Reasoning about Pointers in Refinement Calculus

Ralph-Johan Back
Xiaocong Fan
Viorel Preoteasa

Turku Centre for Computer Science
TUCS Technical Report No 543
June 2003

ISBN 952-12-1198-9
ISSN 1239-1891

Abstract

Pointers are an important programming concept. They are used explicitely or implicitly in many programming languages. In particular, the semantics of object-oriented programming languages rely on pointers. We introduce a semantics for pointer structures. Pointers are seen as indexes and pointer fields are functions from these indexes to values. Using this semantics we turn all pointer operations into simple assignments and then we use refinement calculus techniques to construct a pointer-manipulating program that checks whether or not a single linked list has a loop. We also introduce an induction principle on pointer structures in order to reduce complexity of the proofs.

Keywords: Refinement, Pointer Structures

TUCS Laboratory
Software Construction Laboratory

1 Introduction

Pointers provide an efficient and effective solution to implementing some programming tasks. Moreover, object−oriented languages rely explicitly (e.g. C++, Pascal), or implicitly (e.g. Java, Python, C#, Eiffel) on pointers. However, pointer-manipulating programs are notoriously prone to errors due to pointer dangling, pointer aliasing, null-pointer accessing, and memory leaking. Some languages offer certain mechanisms to prevent the above-mentioned problems from occurring. For instance, the garbage collection mechanism frees the programmer from manually disposing memory and solves the problem of memory leaking. The problem of pointer dangling could be alleviated by always instantiating pointer members of objects with null, and combining it with garbage collection. However, a powerful while relatively simple pointer calculus is highly needed for refining specifications into executables leveraging the flexibility and efficiency of pointers, and for laying a basis for mechanically proving the correctness of existing pointer-manipulating programs using theorem proving systems such as HOL [13], PVS [20], etc.

In this paper, we develop such a formal framework for pointer structures in higher-order logic [10]. The goal of the calculus is to add support to refinement calculus [2] for reasoning about pointer programs. We first introduce a general theory about pointers, where the pointer fields of an object are modeled as functions from objects to objects. The assignment to a pointer field is seen as an update of the corresponding function. We also keep track of all allocated pointers in a subset P of the set of all objects. Allocating a new pointer means updating the set P to include the new element, and disposing an allocated pointer means removing it from P. With the semantics that we propose, all pointer operations become simple assignments, and this enable us to extend the refinement calculus to support pointer and object-oriented constructs.

However, such treatment also brings in complexity in manipulating programs because we are now dealing with functions rather than simple types, which complicates the formulas to be proved. To reduce such complexity, we introduce a principle of induction over the set of pointers accessible from a given starting pointer. Here, we say that a pointer is accessible, only if all the pointers in the corresponding path starting from the initial pointer are allocated.

We then specialize the theory for single linked lists. As an illustrative example, we completely refine a specification for testing whether a single linked list is linear (i.e., leads to null) or has a loop (i.e., the last element points to some element of the list) into a program using an in-place algorithm to reverse the list. In this example, we specify how the algorithm reverses a list by means of a recursive definition. As a result, we get the properties that the loop invariant should satisfy for free. We also prove that by reversing a single linked list twice, we can get the initial list.

2 Related work

There have been many formal treatments of pointer structures; here we concentrate on a few that we found most relevant to our work.

Reynolds [23] describes an axiomatic [14, 12] programming logic for reasoning about correctness of pointer programs. This logic is based on early ideas of Burstall [7], and combines ideas from [19, 22, 15]. He uses a simple imperative programming language with commands for accessing and modifying pointer structures, and for allocation and deallocation of storage. The assertion language is used to express heap properties; in particular, a "separating conjunction" construct express properties that hold for disjoint parts of the heap. Neither the programming language nor the assertion language refers explicitely to the heap. However, as the author notes, the logic is in practice incomplete – new inference rules may be needed for new problems. Moreover, in order to use refinement calculus techniques, we need to refer explicitly to the heap in an assignment specification statement [2]. Contrary to Reynolds's approach, we allow explicit references to the "heap", both in programs and in assertions.

In order to deal with assignments involving pointer variables, Morris [18] generalizes the Hoare axiom of assignment correctness [14] by allowing for pointer fields to be treated as regular program variables. The substitution is done by replacing all aliases of the pointer field with the corresponding expression.

The treatment of pointer-structure fields as global functions from pointers to values can be traced back at least to Burstall [7]. Similar ideas are used prevalently in [16, 4, 6, 9, 17, 5]. Most of these approaches have developed axiomatic semantics for pointer programs.

Butler [8] uses a data refinement mechanism to translate recursive specifications on abstract trees to imperative algorithms on pointer structures. In comparison, we add the induction mechanism directly to the pointer structures, which leads to simpler refinement proofs.

Paige, Ostroff, and Brooke [21] introduce a semantics for reference and expanded types in Eiffel. The theory is expressed in the PVS specification language. References (pointers) are organized in equivalence classes of references to the same object. Creating a new object means creating a new singleton equivalence class that contains a reference to the new object. Assigning to a reference variable means moving this reference from one equivalence class to another. However, in this work we could not see how difficult is to solve a real problem, due to lack of practical examples.

3 Preliminaries

In this section we introduce the programming language we are working with and the refinement rules.

3.1 Data types

We assume that we have a collection of basic data types. Among them we have int, bool, nat, Ω. The set Ω represents the set of all possible pointers (objects). We assume that Ω is an infinite type. We also assume the existence of the function type. The type $A \to B$ denotes the type of all functions from type A to type B. For a type A we denote with $\mathcal{P}_f.A$ the type of all finite subsets of A.

We assume that we have all the usual operations (e.g., $+, -, \leq, \wedge$) defined on int, bool, and nat. We denote with true and false the two values of bool, but also the constant predicates on some type A (functions from A to bool). In addition we also assume that some other operations are available. If A is a type and if p is a predicate on A then we denote by ϵp some arbitrary but fixed element a of A such that $p.a$ is true. If p is false then ϵp is some arbitrary element of A.

If e is an expression of type A where the variable x of type B may occur free, then we denote by $e[x := e']$ the substitution of x with e' in e. We denote by $(\lambda x \bullet e)$ the function that maps $b \in B$ to $e[x := b] \in A$. If $f \in A \to B$ and $a \in A$ then $f.a$ denotes the application of f to a.

For a function f from A to B, $a \in A$ and $b \in B$, we define the update of f in a to b, denoted $f[a \leftarrow b]$, by $(\lambda x : A \bullet \mathit{if}\ x = a\ \mathit{then}\ b\ \mathit{else}\ f.x\ \mathit{fi})$

Lemma 1 *The update of f satisfies the following properties:*

1. $f[x \leftarrow y].x = y$
2. $f[x \leftarrow y][x \leftarrow z] = f[x \leftarrow z]$
3. $f[x \leftarrow f.x] = f$
4. $x \neq z \Rightarrow f[x \leftarrow y].z = f.z$

5. $x \neq y \Rightarrow f[x \leftarrow z][y \leftarrow u] = f[y \leftarrow u][x \leftarrow z]$

Although we use a similar syntax for substitution and update, they are different concepts.

3.2 The programming language

We will use a simple programming language that contains the basic imperative programming constructs. Our language has program variables of data types that we have described so far. The program expressions are built from program variables and constants using the operators that are available for the data types. Although we call it a programming language, this language contains constructs that are not executable. These constructs can be used to specify what the result of the computation should be. This allows us to represent specifications as programs and then refine them to executable programs.

The abstract syntax of the language is given by structural induction. If x, and x' are distinct program variables of the same type, b is a boolean expression, e is a program expression of the same type as x and S, and S' are programs, then the following constructs are programs too:

i.	$\{b\}$	– assertion
ii.	$[\,x := x' \mid b\,]$	– specification assignment
iii.	$\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S'\ \mathsf{fi}$	– if statement
iv.	$\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}$	– while statement
v.	$S\,;S'$	– sequential composition

The statements if, while, and sequential composition are the usual statements that can be found in all imperative programming languages. The assertion $\{b\}$ does nothing if b is true in the current state, and behaves as *abort* (it does not terminate) otherwise. The specification assignment updates the variable x to a value x' that makes b true. If for all x', b is false then it behaves as *magic* (establishes any *postcondition*) [2]. The variable x' is bounded in $[\,x := x' \mid b\,]$.

In order to manipulate programs and prove properties about them we will also need a semantics for them. We use a predicate transformer semantics [11]. The semantics of a program is a function from predicates on states to predicates on states. The intuition of a predicate transformer S applied to a predicate q is the set of the initial states from which the execution of S terminates in a state that satisfies q.

The refinement relation on programs, denoted $\sqsubseteq$, is the pointwise extension of the partial order on predicates over states, i.e. $S \sqsubseteq S'$ if $(\forall q \bullet S.q \subseteq S'.q)$.

The Hoare total-correctness triple [14] $p \;\{\!|\; S \;|\!\}\; q$ is true, if the execution of S from an initial state that satisfies p, is guaranteed to terminate in a state that satisfies q. The intuition behind the refinement relation can be explained in terms of total correctness: S is refined by S', if S' is correct with respect to a precondition p and postcondition q whenever S is.

We define some other programming constructs based on the primitive ones.

1. $\mathsf{skip} = \{\mathsf{true}\}$
2. $(x := e) = [\, x := x' \mid x' = e\,]$ where x' is a variable that does not occur free in e
3. $\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{fi} = \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else\ skip\ fi}$.

If we write the sentences of a program on different lines then we do not use the sequential composition operator. We use indentation to emphasize the body of while or if statements. When indentation is used, we do not use od or fi to end the while and if statements.

3.3 Refinement rules

We list a set of refinement (equivalence) rules that we use in our example. The rules are proven in [2, 3].

assertion introduction If x is not free in e then
$(x := e) = (x := e;\ \{x = e\})$

assertion refinement if $\alpha \Rightarrow \beta$ then $\{\alpha\} \sqsubseteq \{\beta\}$

assignment merge $(x := e\ ;\ x := e') = (x := e'[x := e])$

multiple assignment If x is not free in f then
$(x := e\,; y := f) = (x, y := e, f)$

relational assignment If x' is not free in e then
$[x := x' \,|\, x' = e] = (x := e)$

assignment introduction If $\alpha \Rightarrow \beta[x' := e]$ then
$\{\alpha\}\ ;\ [x := x' \,|\, \beta] \sqsubseteq x := e$

moving assertion $\{x = x_0 \wedge \alpha\}\ ;\ x := e \sqsubseteq$
$x := e\ ;\ \{x = e[x := x_0] \wedge \alpha[x := x_0]\}$

using assertion – assignment If $\alpha \Rightarrow e = e'$ then
$(\{\alpha\}\,;x := e) = (\{\alpha\}\,;x := e')$

adding specification variables If a is not free in S, and S' then $S \sqsubseteq S' \Leftrightarrow$
$(\forall a : \{a = e\}\ ;\ S \sqsubseteq S')$

introducing if statement $\{\alpha\}\ ;\ S \sqsubseteq \mathsf{if}\ \alpha\ \mathsf{then}\ S\ \mathsf{fi}$

unfolding while
$\mathsf{while}\ \alpha\ \mathsf{do}\ S\ \mathsf{od} = \mathsf{if}\ \alpha\ \mathsf{then}\ S\ ;\ \mathsf{while}\ \alpha\ \mathsf{do}\ S\ \mathsf{od}\ \mathsf{fi}$

merge while [3] $\mathsf{while}\ \alpha\ \mathsf{do}\ S\ \mathsf{od}\ =$
$\mathsf{while}\ \alpha \wedge \beta\ \mathsf{do}\ S\ \mathsf{od}\ ;\ \mathsf{while}\ \alpha\ \mathsf{do}\ S\ \mathsf{od}$

using assertion – while If $\alpha \Rightarrow (\beta = \gamma)$ then
$\{\alpha\}\ ;\ \mathsf{while}\ \beta\ \mathsf{do}\ S\ ;\ \{\alpha\}\ \mathsf{od}\ = \{\alpha\}\ ;\ \mathsf{while}\ \gamma\ \mathsf{do}\ S\ ;\ \{\alpha\}\ \mathsf{od}$

while introduction If $\alpha \Rightarrow I$ and $I \wedge \neg\gamma \Rightarrow \beta[x' := x]$ then

```
    $\{\alpha\}\ ;\ [x := x' \,|\, \beta]$
$\sqsubseteq$
    $\{\alpha\}$
    while $\gamma$ do
        $\{I \wedge \gamma\}$
        $[x := x' \,|\, I[x := x'] \wedge t[x := x'] < t]$
        $\{I\}$
    $\{I \wedge \neg\gamma\}$
```

To handle local program variables [1] we assume that we have two programming constructs $\mathsf{add}.x$ and $\mathsf{del}.x$, which add and delete a local program variable x. The two constructs commute with all programs where x does not occur free. We assume that adding x, followed by setting it to some arbitrary value and then deleting it is equivalent to skip. Moreover we assume that the program $\mathsf{add}.x; S; \mathsf{del}.x; \mathsf{add}.x; S'; \mathsf{del}.x$ is refined by $\mathsf{add}.x; S; S'; \mathsf{del}.x$. The following rule can be derived from the properties of add and del.

local variable introduction If x is not free in α and S then
$\{\alpha\}\ ;\ S \sqsubseteq \mathsf{add}.x\ ;\ \{\alpha\}\ ;\ [x := x' \,|\, \beta]\ ;\ S\ ;\ \mathsf{del}.x$

In our example we will omit the statements $\mathsf{add}.x$ and $\mathsf{del}.x$.

4 Dynamic data structures

In this section we show how to capture the basic notions of dynamic data structures (pointers) without any substantial extension of the logical basis as it is presented in [2]. The new programming constructs that we add are defined in terms of the primitives we have already introduced.

A pointer structure declaration is:

$$\textsf{pointer}\ name_i(f_{i,1} : T_{i,1},\ \ldots,\ f_{i,n_i} : T_{i,n_i}) \tag{1}$$

We can have as many pointer structure declarations as we need. The field types $T_{i,1},\ \ldots,\ T_{i,n_i}$ can be any basic types, except Ω, or any pointer structure names that have been already declared or that are going to be declared. For all field types $T_{i,j}$ we define $\textsf{typeof}.T_{i,j}$ by

$$\textsf{typeof}.T_{i,j} = \begin{cases} T_{i,j} & \text{if } T_{i,j} \text{ is a basic type} \\ \Omega & \text{if } T_{i,j} \text{ is pointer structure name} \end{cases}$$

A declaration of the pointer structures $name_1$ to $name_k$ corresponds to the following declarations in terms of basic program constructs.

```
var $name_1, \ldots, name_k : \mathcal{P}_f.\Omega$
var $f_{1,1} : \Omega \rightarrow \textsf{typeof}.T_{1,1}$
...
var $f_{k,n} : \Omega \rightarrow \textsf{typeof}.T_{k,n_k}$
$name_1 := \emptyset$
...
$name_k := \emptyset$                                          (2)
$\textsf{nil} := (\epsilon x : \Omega \bullet \textsf{false})$
new(var p : $\Omega$, var A : $\mathcal{P}_f.\Omega$) :
    $\textsf{p} := (\epsilon x : \Omega \bullet x \neq \textsf{nil} \wedge x \notin \bigcup_i name_i)$
    $\textsf{A} := \textsf{A} \cup \{\textsf{p}\}$
dispose(val p : $\Omega$, var A : $\mathcal{P}_f.\Omega$) :
    $\textsf{A} := \textsf{A} - \{\textsf{p}\}$
```

To allocate and dispose a pointer we have defined the procedures new and dispose. The procedure new has two reference parameters: p for the newly allocated pointer, and A for the set of all allocated pointers of some type. We create a new pointer of type $name_i$ and assign it to p by calling new(p, $name_i$).

The access of a field f of some pointer p, denoted $\textsf{p} \rightarrow f$, is in our case $f.\textsf{p}$. The update of a field $(\textsf{p} \rightarrow f := \textsf{q})$ is $f := f[\textsf{p} \leftarrow \textsf{q}]$.

With these definitions all pointer operations becomes simple assignments and we can use the assignment refinement rules for them.

For the rest of this section we assume that we have a program that declares the pointer structures $name_1$, ..., $name_k$. For any pointer structure field $f_{i,j}$ we denote by $a_{i,j}$ the tuple ($f_{i,j}$, $name_i$, $T_{i,j}$). We use the notation $a_{i,j}$ for both, the tuple, and its first component. We refer to the second element of $a_{i,j}$ by $\mathsf{dom}.a_{i,j}$ and to the third by $\mathsf{range}.a_{i,j}$. We denote by A the set of all $a_{i,j}$ for which $T_{i,j}$ is a pointer structure name, and by A^* the free monoid generated by A. We denote with 1 the empty word. For $\alpha, \beta \in A^*$ we denote $\alpha \leq \beta$ iff α is a prefix of β and $\alpha < \beta$ iff α is a proper prefix of β. If $\alpha \leq \beta$, then we denote with $\alpha^{-1}\beta$ the word obtained from β by removing the prefix α, i.e. $\alpha^{-1}\beta = \gamma$ where γ is such that $\beta = \alpha\gamma$.

Let pstr be $\bigcup_i name_i$.

Definition 2 *For $\alpha \in A^*$ and $\mathsf{p} \in \Omega$. We define $\alpha.\mathsf{p}$ by induction on α:*

$$1.\mathsf{p} = \mathsf{p} \textit{ and } a\alpha.\mathsf{p} = \alpha.(a.\mathsf{p}).$$

A straightforward consequence of the above definition is that if $\alpha, \beta \in A^*$ and $\mathsf{p} \in \Omega$ then $\alpha\beta.\mathsf{p} = \beta.(\alpha.\mathsf{p})$.

We define $\mathsf{p} \overset{\alpha}{\underset{A}{\longmapsto}} \mathsf{q}$ to be true if we can reach the pointer q from p following the path α by accessing only proper (allocated) pointers. Formally:

Definition 3 *If $\mathsf{p}, \mathsf{q} \in \Omega$, $a \in A$, and $\alpha \in A^*$ then*

1. $\mathsf{p} \overset{1}{\underset{A}{\longmapsto}} \mathsf{q}$ *if* $\mathsf{p} = \mathsf{q}$ *and* $\mathsf{p} \in \mathsf{pstr} \cup \{\mathsf{nil}\}$
2. $\mathsf{p} \overset{a}{\underset{A}{\longmapsto}} \mathsf{q}$ *if* $a.\mathsf{p} = \mathsf{q} \land \mathsf{p} \in \mathsf{dom}.a \ \land \mathsf{q} \in \mathsf{range}.a \cup \{\mathsf{nil}\}$
3. $\mathsf{p} \overset{a\alpha}{\underset{A}{\longmapsto}} \mathsf{q}$ *if* $(\exists \mathsf{r} \in \Omega \bullet \mathsf{p} \overset{a}{\underset{A}{\longmapsto}} \mathsf{r} \land \mathsf{r} \overset{\alpha}{\underset{A}{\longmapsto}} \mathsf{q})$

When the set A is fixed, we will omit it from the notation $\mathsf{p} \overset{\alpha}{\underset{A}{\longmapsto}} \mathsf{q}$ and in general from any notation that has A as a parameter.

Definition 4 *Let* $[\mathsf{p}]_A = \{\mathsf{q} \mid (\exists \alpha \in A^* \bullet \mathsf{p} \overset{\alpha}{\underset{A}{\longmapsto}} \mathsf{q}) \land \ \mathsf{q} \neq \mathsf{nil}\}$ *and* $|\mathsf{p}|_A = |\ [\mathsf{p}]_A|$

Lemma 5 *If $\alpha, \beta \in A^*$ then*

1. $\mathsf{p} \xrightarrow{\alpha} \mathsf{q} \wedge \mathsf{p} \xrightarrow{\alpha} \mathsf{r} \Rightarrow \mathsf{q} = \mathsf{r} = \alpha.\mathsf{p}$
2. $\mathsf{q} \xrightarrow{\alpha\beta} \mathsf{p} \Leftrightarrow (\exists \mathsf{r} \bullet \mathsf{q} \xrightarrow{\alpha} \mathsf{r} \xrightarrow{\beta} \mathsf{p})$
3. $\mathsf{q} \xrightarrow{\alpha} \mathsf{p} \wedge \mathsf{q} \xrightarrow{\beta} \mathsf{r} \wedge \alpha \leq \beta \Rightarrow \mathsf{p} \xrightarrow{\alpha^{-1}\beta} \mathsf{r}$
4. $\mathsf{p} \in \mathsf{pstr} \Leftrightarrow \mathsf{p} \in [\mathsf{p}]_A$
5. $\mathsf{p} \in \mathsf{pstr} \Leftrightarrow [\mathsf{p}]_A \neq \emptyset$

Theorem 6 (Pointer Induction) *If P is a predicate on Ω then*

$$P.\mathsf{p} \wedge (\forall \mathsf{q} \in [\mathsf{p}], a \in A, \mathsf{r} \in \Omega - \{\mathsf{nil}\} \bullet$$
$$P.\mathsf{q} \wedge \mathsf{q} \xrightarrow{a} \mathsf{r} \Rightarrow P.\mathsf{r}) \Rightarrow (\forall \mathsf{q} \in [\mathsf{p}] \bullet P.\mathsf{q})$$

Proof. If $\mathsf{q} \in [\mathsf{p}]$ then exists $\alpha \in [\mathsf{p}]$ such that $\mathsf{p} \xrightarrow[A]{\alpha} \mathsf{q}$ and $\mathsf{q} \neq \mathsf{nil}$. We can prove $P.\mathsf{q}$ by induction on the length of α.

5 Single linked list

Single linked lists are pointer structures that contain an info field of some type (integer in our example) and a next field that gives for a pointer the next element in a list.

$$\mathsf{pointer\ plist\ (info : int,\ next : plist)}$$

In the case of only one link ($A = \{(\mathsf{next},\ \mathsf{plist},\ \mathsf{plist})\}$), the set A^* is isomorphic with the set of natural numbers. We will use the notation $\mathsf{p} \xrightarrow[A]{n} \mathsf{q}$ instead of $\mathsf{p} \xrightarrow[A]{\mathsf{next}^n} \mathsf{q}$. The fact $\mathsf{next}^n \leq \mathsf{next}^m$ becomes $n \leq m$ and $(\mathsf{next}^n)^{-1}\mathsf{next}^m$ becomes $m - n$. We will mention instead of the set index A just the components of A that changes. For example if we have two next functions next and next_0 we write $\mathsf{p} \xrightarrow[\mathsf{next}]{n} \mathsf{q}$, and $\mathsf{p} \xrightarrow[\mathsf{next}_0]{n} \mathsf{q}$ instead of $\mathsf{p} \xrightarrow[A]{n} \mathsf{q}$ and $\mathsf{p} \xrightarrow[A_0]{n} \mathsf{q}$ where $A_0 = \{(\mathsf{next}_0,\ \mathsf{plist},\ \mathsf{plist})\}$

Lemma 7 $\mathsf{p} \in [\mathsf{q}]$ *if and only if there is an unique $i < |\mathsf{q}|$ such that* $\mathsf{q} \xrightarrow{i} \mathsf{p}$.

Corollary 8 $[\mathsf{p}] = \{\mathsf{next}^i.\mathsf{p} \mid i < |\mathsf{p}|\}$. *If $i, j < |\mathsf{p}|$ and $i \neq j$ then* $\mathsf{next}^i.\mathsf{p} \neq \mathsf{next}^j.\mathsf{p}$.

Definition 9 *If* $n = |\mathsf{p}|_A$ *then we define* $\mathsf{last}_A.\mathsf{p} \in \mathsf{plist}$ *by* $\mathsf{last}_A.\mathsf{p} = \mathsf{next}^{n-1}.\mathsf{p}$.

Definition 10 *If* $\mathsf{q} \in [\mathsf{p}]_A$ *then we define*

i. $[\mathsf{p} : \mathsf{q}]_A = \{\mathsf{s} \mid \exists i, j : \mathsf{p} \overset{i}{\underset{A}{\longmapsto}} \mathsf{s} \overset{j}{\underset{A}{\longmapsto}} \mathsf{q} \wedge i + j < |\mathsf{p}|_A\}$

ii. $[\mathsf{p} : \mathsf{q})_A = \{\mathsf{s} \mid \exists i, j : \mathsf{p} \overset{i}{\underset{A}{\longmapsto}} \mathsf{s} \overset{j}{\underset{A}{\longmapsto}} \mathsf{q} \wedge i + j < |\mathsf{p}|_A \wedge 0 < j\}$

iii. $(\mathsf{p} : \mathsf{q}]_A = \{\mathsf{s} \mid \exists i, j : \mathsf{p} \overset{i}{\underset{A}{\longmapsto}} \mathsf{s} \overset{j}{\underset{A}{\longmapsto}} \mathsf{q} \wedge i + j < |\mathsf{p}|_A \wedge 0 < i\}$

iv. $(\mathsf{p} : \mathsf{q})_A = \{\mathsf{s} \mid \exists i, j : \mathsf{p} \overset{i}{\underset{A}{\longmapsto}} \mathsf{s} \overset{j}{\underset{A}{\longmapsto}} \mathsf{q} \wedge i + j < |\mathsf{p}|_A \wedge 0 < i, j\}$

Lemma 11 *If* $\mathsf{q} \in [\mathsf{p}]$ *then*

i. $[\mathsf{p}] = [\mathsf{p} : \mathsf{last}.\mathsf{p}]$

ii. $[\mathsf{p}] = [\mathsf{p} : \mathsf{q}) \cup [\mathsf{q}]$

iii. $\mathsf{s} \in [\mathsf{p} : \mathsf{q}] \Rightarrow [\mathsf{p} : \mathsf{q}] = [\mathsf{p} : \mathsf{s}) + \{\mathsf{s}\} + (\mathsf{s} : \mathsf{q}]$

iv. $\mathsf{s} \in [\mathsf{p} : \mathsf{q}) \Rightarrow \mathsf{next}.\mathsf{s} \in (\mathsf{p} : \mathsf{q}] \wedge \mathsf{next}.\mathsf{s} \notin [\mathsf{p} : \mathsf{s}]$

v. $\mathsf{p} \neq \mathsf{q} \Rightarrow [\mathsf{p} : \mathsf{q}] = \{\mathsf{p}\} + (\mathsf{p} : \mathsf{q}) + \{\mathsf{q}\}$

Where $x = y + z$ *denotes the fact that* $x = y \cup z$ *and* $y \cap z = \emptyset$.

Proof. Using Corollary 8.

Definition 12 *If* $\mathsf{q} \in [\mathsf{p}]_A$ *then we define* $|\mathsf{p} : \mathsf{q}|_A = |[\mathsf{p} : \mathsf{q}]_A|$

Lemma 13 *If* $\mathsf{q} \in [\mathsf{p}]$ *and* $n = |\mathsf{p} : \mathsf{q}|$ *then* $\mathsf{p} \overset{n-1}{\longmapsto} \mathsf{q}$.

Theorem 14 (Length decreasing) *If* $\mathsf{q} \in [\mathsf{p}]$ *and* $\mathsf{s} \in [\mathsf{p} : \mathsf{q})$ *then* $|\mathsf{s} : \mathsf{q}| = |\mathsf{next}.\mathsf{s} : \mathsf{q}| + 1$.

Theorem 15 (List induction) *If* P *is a predicate on* Ω *and* $\mathsf{q} \in [\mathsf{p}]$ *then*

$$P.\mathsf{p} \wedge (\forall \mathsf{r} \in [\mathsf{p} : \mathsf{q}) \bullet P.\mathsf{r} \Rightarrow P.(\mathsf{next}.\mathsf{r})) \Rightarrow (\forall \mathsf{r} \in [\mathsf{p} : \mathsf{q}] \bullet P.\mathsf{r})$$

Proof. By induction on $|\mathsf{p} : \mathsf{r}|$.

In the case of lists we can also introduce a principle of definition by induction. In order to define a function f on $[\mathsf{p} : \mathsf{q}]$ it is enough to define $f.\mathsf{p}$, and for all $\mathsf{r} \in [\mathsf{p} : \mathsf{q})$ to define $f.(\mathsf{next.r})$ assuming that $f.\mathsf{r}$ is defined.

Definition 16 *For all* $\mathsf{p} \in \mathsf{plist}$ *we define* $\mathsf{linear}_A.\mathsf{p}$, $\mathsf{circular}_A.\mathsf{p}$, $\mathsf{loop}_A.\mathsf{p}$, *and* $\mathsf{list}_A.\mathsf{p} \in \mathsf{bool}$ *by:*

$$\begin{aligned}
\mathsf{linear}_A.\mathsf{p} &= (\mathsf{next}.(\mathsf{last}_A.\mathsf{p}) = \mathsf{nil}) \\
\mathsf{circular}_A.\mathsf{p} &= (\mathsf{next}.(\mathsf{last}_A.\mathsf{p}) = \mathsf{p}) \\
\mathsf{loop}_A.\mathsf{p} &= (\mathsf{next}.(\mathsf{last}_A.\mathsf{p}) \in [\mathsf{p}]_A) \\
\mathsf{list}_A.\mathsf{p} &= \mathsf{linear}_A.\mathsf{p} \lor \mathsf{loop}_A.\mathsf{p}
\end{aligned}$$

Lemma 17 *If* $\mathsf{linear.p}$ *then* $\mathsf{p} \xmapsto{|\mathsf{p}|} \mathsf{nil}$.

Lemma 18 *If* $\mathsf{linear.p}$ *and* $\mathsf{q} \in [\mathsf{p}]$ *then* $\mathsf{q} = \mathsf{last.p} \Leftrightarrow \mathsf{next.q} = \mathsf{nil}$.

Lemma 19 *If* $\mathsf{circular.p}$ *then* $\mathsf{circular.(next.p)}$, $[\mathsf{p}] = [\mathsf{next.p}]$, *and* $\mathsf{last.(next.p)} = \mathsf{p}$.

Lemma 20 *If* $\mathsf{loop.p}$ *then* $\mathsf{circular.(next.(last.p))}$.

5.1 Partial reverse of a list

We define for the pointers $\mathsf{p} \in \mathsf{plist}$, $\mathsf{q} \in [\mathsf{p}]$, and $\mathsf{e} \in \Omega$ a partial reverse of the list from p to q as in Figure 1. The links from p to $\mathsf{next.q}$ (the arrows labeled with 1 in Figure 1) are replaced by links from q to p (the dashed arrows in Figure 1). We also create a link from p to e. In different contexts the pointer e will play different roles. For example, to reverse a linear list we partially reverse it until the last element and use nil as e.

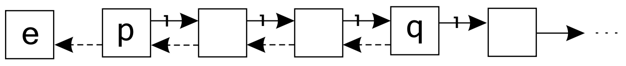

Figure 1: Partial reverse of a list

Definition 21 *Suppose that* $\mathsf{q} \in [\mathsf{p}]$ *and* $\mathsf{e} \in \Omega$. *We define the function* $\mathsf{preverse.next.p.q.e}$ *of type* $\Omega \to \Omega$ *by induction on* $\mathsf{r} \in [\mathsf{p} : \mathsf{q}]_{\mathsf{next}}$.

- *Case* $\mathsf{r} = \mathsf{p}$*:*
  $\mathsf{preverse.next.p.r.e} = \mathsf{next}[\mathsf{p} \leftarrow \mathsf{e}]$

- *Case* $\mathsf{r} \in [\mathsf{p} : \mathsf{q})_{\mathsf{next}}$*:*
  $\mathsf{preverse.next.p.(next.r).e} =$
  $\quad (\mathsf{preverse.next.p.r.e})[\mathsf{next.r} \leftarrow \mathsf{r}]$

When next, p, q and e are fixed we denote with $f.\mathsf{r} = \mathsf{preverse.next.p.r.e}$ for all $\mathsf{r} \in [\mathsf{p} : \mathsf{q}]_{\mathsf{next}}$ and $\mathsf{next}_0 = f.\mathsf{q}$.

Lemma 22 (Partial reverse – properties) *If* $\mathsf{q} \in [\mathsf{p}]_{\mathsf{next}}$ *then*

1. $\forall \mathsf{r} \in [\mathsf{p} : \mathsf{q}]_{\mathsf{next}} \bullet (f.\mathsf{r}).\mathsf{p} = \mathsf{e} \wedge \mathsf{p} \in [\mathsf{r}]_{f.\mathsf{r}} \ \wedge [\mathsf{p} : \mathsf{r}]_{\mathsf{next}} = [\mathsf{r} : \mathsf{p}]_{f.\mathsf{r}}$

2. $\forall \mathsf{r} \in [\mathsf{p} : \mathsf{q}]_{\mathsf{next}} \bullet \mathsf{next}_0.\mathsf{r} = (f.\mathsf{r}).\mathsf{r}$

3. $\forall \mathsf{r} \in [\mathsf{p} : \mathsf{q})_{\mathsf{next}} \bullet \mathsf{next}_0.(\mathsf{next.r}) = \mathsf{r}$

4. $\forall \mathsf{r} \notin [\mathsf{p} : \mathsf{q}]_{\mathsf{next}} \bullet \mathsf{next}_0.\mathsf{r} = \mathsf{next.r}$

Proof. By list induction.

Lemma 23 (Partial reverse – commutativity) *If* $\mathsf{q} \in [\mathsf{p}]_{\mathsf{next}}$ *and* $\mathsf{q}' \in [\mathsf{p}']_{\mathsf{next}}$ *such that* $[\mathsf{p} : \mathsf{q}]_{\mathsf{next}} \cap [\mathsf{p}' : \mathsf{q}']_{\mathsf{next}} = \emptyset$ *then*

1. $\forall \mathsf{s} \notin [\mathsf{p} : \mathsf{q}]_{\mathsf{next}} \mathsf{preverse.}(\mathsf{next}[\mathsf{s} \leftarrow \mathsf{e}]).\mathsf{p.q.e}' =$
   $\quad (\mathsf{preverse.next.p.q.e}')[\mathsf{s} \leftarrow \mathsf{e}]$

2. $\mathsf{preverse.(preverse.next.p.q.e).p'.q'.e'} =$
   $\quad \mathsf{preverse.(preverse.next.p'.q'.e').p.q.e}$

Lemma 24 (Partial reverse – split) *If* $\mathsf{q} \in [\mathsf{p}]_{\mathsf{next}}$ *then*

$$\forall \mathsf{r} \in [\mathsf{p} : \mathsf{q})_{\mathsf{next}} \mathsf{preverse.next.p.q.e} = \mathsf{preverse.(preverse.next.p.r.e).(next.r).q.r}$$

Proof. If $[\mathsf{p} : \mathsf{q})_{\mathsf{next}}$ is empty there is nothing to prove. Otherwise there exists $\mathsf{q}_0 \in [\mathsf{p} : \mathsf{q})_{\mathsf{next}}$ such that $[\mathsf{p} : \mathsf{q})_{\mathsf{next}} = [\mathsf{p} : \mathsf{q}_0]_{\mathsf{next}}$. We prove the property above by induction on $\mathsf{r} \in [\mathsf{p} : \mathsf{q}']_{\mathsf{next}}$.

Lemma 25 (Reverse twice) *If* $\mathsf{q} \in [\mathsf{p}]_{\mathsf{next}}$ *then*

$$\mathsf{preverse.(preverse.next.p.q.e).q.p.(next.q)} = \mathsf{next}$$

Proof. We prove by induction on $\mathsf{r} \in [\mathsf{p} : \mathsf{q}]_{\mathsf{next}}$ that

$$\mathsf{preverse.(preverse.next.p.r.e).r.p.(next.r)} = \mathsf{next}$$

- Case $\mathsf{r} = \mathsf{p}$:

$$\begin{aligned}
&\mathsf{preverse.(preverse.next.p.p.e).p.p.(next.p)}\\
= &\ \{\mathsf{preverse}\ \text{definition}\}\\
&\mathsf{next}[\mathsf{p} \leftarrow \mathsf{e}][\mathsf{p} \leftarrow \mathsf{next.p}]\\
= &\ \{\text{Lemma 1}\}\\
&\mathsf{next}
\end{aligned}$$

- Case $\mathsf{r} \in [\mathsf{p} : \mathsf{q})_{\mathsf{next}}$, assume $\mathsf{preverse.(preverse.next.p.r.e).r.p.(next.r)} = \mathsf{next}$ and denote $\mathsf{next}_0 = \mathsf{preverse.next.p.r.e}$ and $\mathsf{next}_1 = \mathsf{preverse.next.p.(next.r).e} = \mathsf{next}_0[\mathsf{next.r} \leftarrow \mathsf{r}]$

$$\begin{aligned}
&\mathsf{preverse.next}_1\mathsf{.(next.r).p.(next.(next.r))}\\
= &\ \{\text{Lemmas 23 and 24}\}\\
&\mathsf{preverse.next}_1\mathsf{.(next}_1\mathsf{.(next.r)).p.(next.r)}[\mathsf{next.r} \leftarrow \mathsf{next.(next.r)}]\\
= &\ \{\text{assumptions}\}\\
&\mathsf{preverse.next}_1\mathsf{.r.p.(next.r)}[\mathsf{next.r} \leftarrow \mathsf{next.(next.r)}]\\
= &\ \{\text{assumptions}\}\\
&\mathsf{preverse.(next}_0[\mathsf{next.r} \leftarrow \mathsf{r}]\mathsf{).r.p.(next.r)}[\mathsf{next.r} \leftarrow \mathsf{next.(next.r)}]\\
= &\ \{\text{Lemma 23}\}\\
&\mathsf{preverse.next}_0\mathsf{.r.p.(next.r)}[\mathsf{next.r} \leftarrow \mathsf{r}][\mathsf{next.r} \leftarrow \mathsf{next.(next.r)}]\\
= &\ \{\text{assumptions and Lemma 1}\}\\
&\mathsf{next}
\end{aligned}$$

Lemma 26 *If* $\mathsf{q} \in [\mathsf{p}]_{\mathsf{next}}$ *and* $\mathsf{next}_0 = \mathsf{preverse.next.p.q.nil}$ *then* $\mathsf{linear}_{\mathsf{next}_0}\mathsf{.q}$ *and* $\mathsf{p} = \mathsf{last}_{\mathsf{next}_0}\mathsf{.q}$

5.2 Linear lists

To reverse a linear list we have to partially reverse the list from the head to the last element. We also have to end the reversed list with nil. The head of the resulting list is the last element of the initial list. Formally we have.

Definition 27 *If* $\mathsf{linear}_{\mathsf{next}}\mathsf{.p}$ *then we define* $\mathsf{reverse.(next,\ p)}$ *given by*

$$\mathsf{reverse.(next,\ p)} = (\mathsf{preverse.next.p.(last}_{\mathsf{next}}\mathsf{.p).nil},\ \mathsf{last}_{\mathsf{next}}\mathsf{.p})$$

Theorem 28 *If* $\mathsf{linear}_{\mathsf{next}}\mathsf{.p}$ *and* $(\mathsf{next}_0,\ \mathsf{p}_0) = \mathsf{reverse.(next,\ p)}$ *then* *i.* $\mathsf{linear}_{\mathsf{next}_0}.\mathsf{p}_0$

ii. $[\mathsf{p}_0]_{\mathsf{next}_0} = [\mathsf{p}]_{\mathsf{next}}$

iii. $\mathsf{last}_{\mathsf{next}_0}.\mathsf{p}_0 = \mathsf{p}$

iv. *if* $|\mathsf{p}|_{\mathsf{next}} > 1$ *then* $\mathsf{p} \neq \mathsf{p}_0$

v. $\mathsf{reverse}^2.(\mathsf{next},\ \mathsf{p}) = (\mathsf{next},\ \mathsf{p})$

The Theorem 28 states that if we reverse a linear list twice we get the original list. It also says that if the list has at least two elements then the head of the resulting list is different from the head of the original one. This fact will be used in the final algorithm that decides whether the list has a loop or not. The algorithm uses the fact that when reversing a loop list, the new list has the same header as the original one.

5.3 Loop lists

Reversing a loop list is equivalent to reversing the circular part of it. In Figure 2 we replace the arrows labeled by 1 with dashed arrows. This, in turn, is equivalent to partially reversing the list from h to q using q as e.

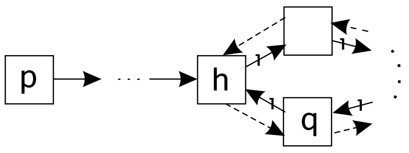

Figure 2: Reverse of a loop list

Definition 29 *If* $\mathsf{loop}_{\mathsf{next}}.\mathsf{p}$, $\mathsf{q} = \mathsf{last}_{\mathsf{next}}.\mathsf{p}$ *and* $\mathsf{h} = \mathsf{next}.\mathsf{q}$ *then we define* $\mathsf{reverse}.(\mathsf{next},\ \mathsf{p})$ *given by*

$$\mathsf{reverse}.(\mathsf{next},\ \mathsf{p}) = (\mathsf{preverse}.\mathsf{next}.\mathsf{h}.\mathsf{q}.\mathsf{q},\ \mathsf{p})$$

Before giving the main theorem about the properties satisfied by the reverse of a loop list we give some results about reversing a circular list.

Lemma 30 *If* $\mathsf{circular}_{\mathsf{next}}.\mathsf{p}$, $\mathsf{q} = \mathsf{last}_{\mathsf{next}}.\mathsf{p}$ *then*

$$\mathsf{preverse}.\mathsf{next}.\mathsf{p}.\mathsf{q}.\mathsf{q} = \mathsf{preverse}.\mathsf{next}.(\mathsf{next}.\mathsf{p}).\mathsf{p}.\mathsf{p}$$

Lemma 31 *If* $\mathsf{circular}_{\mathsf{next}}.\mathsf{p}$, $\mathsf{q} = \mathsf{last}_{\mathsf{next}}.\mathsf{p}$ *and* $\mathsf{next}_0 = \mathsf{preverse.next.p.q.q}$ *then* $\mathsf{circular}_{\mathsf{next}_0}.\mathsf{p}$ *and* $\mathsf{p} = \mathsf{last}_{\mathsf{next}_0}.\mathsf{q}$

Theorem 32 *If* $\mathsf{loop}_{\mathsf{next}}.\mathsf{p}$, $\mathsf{h} = \mathsf{next}.(\mathsf{last}_{\mathsf{next}}.\mathsf{p})$ *and* $(\mathsf{next}_0,\ \mathsf{p}_0) = \mathsf{reverse}.(\mathsf{next},\ \mathsf{p})$, *then*

i. $\mathsf{p} = \mathsf{p}_0$

ii. $\mathsf{loop}_{\mathsf{next}_0}.\mathsf{p}_0$

iii. $[\mathsf{p}_0]_{\mathsf{next}_0} = [\mathsf{p}]_{\mathsf{next}}$

iv. $\mathsf{last}_{\mathsf{next}_0}.\mathsf{p}_0 = \mathsf{next.h}$

v. $\mathsf{reverse}^2.(\mathsf{next},\ \mathsf{p}) = (\mathsf{next},\ \mathsf{p})$

Proof. Using Lemmas 31, 30, and 25

Although the definition of reverse for a loop list is sufficient for reasoning about its properties, it is not good for implementation purposes. The pointer h is not known when the program starts. We do not even know whether we have a loop list. In the next theorem we show that reversing a loop list is equivalent to reversing the elements from p to q (reversing the arrows labeled with 1 in Figure 3) and then reversing back the elements from h to p (reversing the arrows labeled with 2). The final reversed list is given by the dashed arrows in Figure 3.

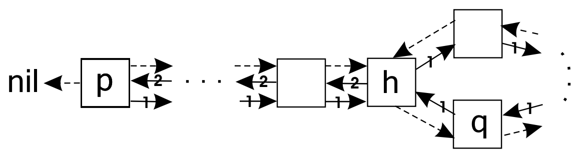

Figure 3: Compute reverse of a loop list

Theorem 33 (Compute reverse of a loop list)
If $\mathsf{loop}_{\mathsf{next}}.\mathsf{p}$, $\mathsf{q} = \mathsf{last}_{\mathsf{next}}.\mathsf{p}$, $\mathsf{h} = \mathsf{next.q}$ *and* $\mathsf{next}_0 = \mathsf{preverse.next.p.q.nil}$ *then*

i. $\mathsf{linear}_{\mathsf{next}_0}.\mathsf{h}$

ii. $\mathsf{p} = \mathsf{last}_{\mathsf{next}_0}.\mathsf{h}$

iii. $\mathsf{preverse.next.h.q.q} = \mathsf{preverse.next}_0.\mathsf{h.p.q}$

Proof. We prove the case $\mathsf{p} \neq \mathsf{h}$. It follows that exists $\mathsf{h}_0 \in [\mathsf{p} : \mathsf{h})_{\mathsf{next}}$ such that $\mathsf{next}.\mathsf{h}_0 = \mathsf{h}$. It follows $\mathsf{next}_0.\mathsf{h}_0 = \mathsf{h}$.

$$
\begin{aligned}
& \mathsf{preverse.next_0.h.p.q} \\
= & \ \{\text{Lemmas 23 and 24}\} \\
& \mathsf{preverse.next_0.(next_0.h).p.h[h \leftarrow q]} \\
= & \ \{\text{assumptions}\} \\
& \mathsf{preverse.(preverse.next.p.q.nil).h_0.p.h[h \leftarrow q]} \\
= & \ \{\text{Lemma 24}\} \\
& \mathsf{preverse.(preverse.(preverse.next.p.h_0.nil).(next.h_0).q.h_0).h_0.p.h[h \leftarrow q]} \\
= & \ \{\text{Lemma 23}\} \\
& \mathsf{preverse.(preverse.(preverse.next.p.h_0.nil).h_0.p.h).(next.h_0).q.h_0[h \leftarrow q]} \\
= & \ \{\text{assumptions}\} \\
& \mathsf{preverse.(preverse.(preverse.next.p.h_0.nil).h_0.p.(next.h_0)).h.q.h_0[h \leftarrow q]} \\
= & \ \{\text{Lemma 25}\} \\
& \mathsf{preverse.next.h.q.h_0[h \leftarrow q]} \\
= & \ \{\text{Lemmas 23 and 24}\} \\
& \mathsf{preverse.next.h.q.q}
\end{aligned}
$$

5.4 Refining a program for checking if a list is linear or not

We first refine the partial reverse of a list to a while program. Using this refinement then we refine the reversing of a linear and a loop list to the same program. Finally we write a program that tests whether or not a list has a loop and prove that it is correct.

Lemma 34 (Refinement of preverse) *If α is a formula that does not contain the variables* next, s, r *free then for all program expressions* q *that does not contain* next, s, r *free we have*

```
    $\{\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{q} \in [\mathsf{p}]_{\mathsf{next}_0} \wedge \alpha\}$
    $\mathsf{next,\ s := preverse.next_0.p.q.e,\ q}$

$\sqsubseteq$

    $\mathsf{next,\ s,\ r := next[p \leftarrow e],\ p,\ next[p]}$
    $\{\mathsf{q} \in [\mathsf{p}]_{\mathsf{next}_0} \wedge \mathsf{s} \in [\mathsf{p} : \mathsf{q}]_{\mathsf{next}_0} \wedge \mathsf{r} = \mathsf{next}_0.\mathsf{s} \wedge \alpha\}$
    $\mathsf{while\ s \neq q\ \ do}$
        $\mathsf{next,\ s,\ r := next[r \leftarrow s],\ r,\ next.r}$
        $\{\mathsf{q} \in [\mathsf{p}]_{\mathsf{next}_0} \wedge \mathsf{s} \in [\mathsf{p} : \mathsf{q}]_{\mathsf{next}_0} \wedge \mathsf{r} = \mathsf{next}_0.\mathsf{s} \wedge \alpha\}$
    $\{\mathsf{s} = \mathsf{q} \wedge \mathsf{q} \in [\mathsf{p}]_{\mathsf{next}_0} \wedge \mathsf{r} = \mathsf{next}_0.\mathsf{q} \wedge \alpha\}$
```

Moreover if $\mathsf{linear}_{\mathsf{next}_0}.\mathsf{p}$ *and* $\mathsf{q} = \mathsf{last}_{\mathsf{next}_0}.\mathsf{p}$ *then the* while *condition can be replaced by* $\mathsf{r} \neq \mathsf{nil}$.

Proof:

```
    $\{\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{q} \in [\mathsf{p}]_{\mathsf{next}_0} \wedge \alpha\}$
    $\mathsf{next,\ s := preverse.next_0.p.q.e,\ q}$

$\sqsubseteq$ {local variable introduction}

    $\{\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{q} \in [\mathsf{p}]_{\mathsf{next}_0} \wedge \alpha\}$
    $[\mathsf{next,\ s,\ r := next',\ s',\ r' \mid next' = preverse.next_0.p.q.e \wedge s' = s}]$

$\sqsubseteq$ {assignment merge}

    $\{\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{q} \in [\mathsf{p}]_{\mathsf{next}_0} \wedge \alpha\}$
    $\mathsf{next,\ s,\ r := next_0[p \leftarrow e],\ p,\ next_0.p}$
    $[\mathsf{next,\ s,\ r := next',\ s',\ r' \mid next' = preverse.next_0.p.q.e \wedge s' = s}]$

$\sqsubseteq$ {moving assertion}

    $\mathsf{next,\ s,\ r := next[p \leftarrow e],\ p,\ next.p}$
    $\{\mathsf{next} = \mathsf{next}_0[\mathsf{p} \leftarrow \mathsf{e}] \wedge \mathsf{q} \in [\mathsf{p}]_{\mathsf{next}_0} \wedge \mathsf{s} = \mathsf{p} \wedge \mathsf{r} = \mathsf{next}_0.\mathsf{s} \wedge \alpha\}$
    $[\mathsf{next,\ s,\ r := next',\ s',\ r' \mid next' = preverse.next_0.p.q.e \wedge s' = s}]$

$\sqsubseteq$ {while introduction}
```

- Let $I = \mathsf{q} \in [\mathsf{p}]_{\mathsf{next}_0} \wedge \mathsf{s} \in [\mathsf{p} : \mathsf{q}]_{\mathsf{next}_0} \wedge \mathsf{r} = \mathsf{next}_0.\mathsf{s} \wedge$
  $\mathsf{next} = \mathsf{preverse.next}_0.\mathsf{p.s.e} \wedge \alpha$
- Let $t = |\mathsf{s} : \mathsf{q}|_{\mathsf{next}_0}$
- $\mathsf{next} = \mathsf{next}_0[\mathsf{p} \leftarrow \mathsf{e}] \wedge \mathsf{q} \in [\mathsf{p}]_{\mathsf{next}_0} \wedge \mathsf{s} = \mathsf{p} \wedge$
  $\mathsf{r} = \mathsf{next}_0.\mathsf{s} \wedge \alpha \Rightarrow I$

- $I \wedge \mathsf{s} = \mathsf{q} \Rightarrow \mathsf{next} = \mathsf{preverse.next_0.p.q.e} \wedge \mathsf{q} = \mathsf{s}$

```
$\mathsf{next,\ s,\ r := next[p \leftarrow e],\ p,\ next.p}$
$\{I\}$
$\mathsf{while\ s \neq q\ \ do}$
    $\{I \wedge \mathsf{s} \neq \mathsf{q}\}$
    $[\mathsf{next,\ s,\ r := next',\ s',\ r'} \mid I(\mathsf{next', s', r'}) \wedge t(\mathsf{next', s', r'}) < t]$
    $\{I\}$
$\{I \wedge \mathsf{s} = \mathsf{q}\}$
```

$\sqsubseteq$ {assignment introduction}

- $I \wedge \mathsf{s} \neq \mathsf{q} \Rightarrow I(\mathsf{next[r \leftarrow s], r, next.r}) \wedge$
  $t(\mathsf{next[r \leftarrow s], r, next.r}) < t$

```
$\mathsf{next,\ s,\ r := next[p \leftarrow e],\ p,\ next.p}$
$\{I\}$
$\mathsf{while\ s \neq q\ \ do}$
    $\mathsf{next,\ s,\ r := next[r \leftarrow s],\ r,\ next.r}$
    $\{I\}$
$\{I \wedge \mathsf{s} = \mathsf{q}\}$
```

$\sqsubseteq$ {assertion refinement}

```
$\mathsf{next,\ s,\ r := next[p \leftarrow e],\ q,\ next[p]}$
$\{\mathsf{q} \in [\mathsf{p}]_{\mathsf{next_0}} \wedge \mathsf{s} \in [\mathsf{p : q}]_{\mathsf{next_0}} \wedge \mathsf{r} = \mathsf{next_0.s} \wedge \alpha\}$
$\mathsf{while\ s \neq q\ \ do}$
    $\mathsf{next,\ s,\ r := next[r \leftarrow s],\ r,\ next.r}$
    $\{\mathsf{q} \in [\mathsf{p}]_{\mathsf{next_0}} \wedge \mathsf{s} \in [\mathsf{p : q}]_{\mathsf{next_0}} \wedge \mathsf{r} = \mathsf{next_0.s} \wedge \alpha\}$
$\{\mathsf{s} = \mathsf{q} \wedge \mathsf{q} \in [\mathsf{p}]_{\mathsf{next_0}} \wedge \mathsf{r} = \mathsf{next_0.q} \wedge \alpha\}$
```

Lemma 35 (Refinement of reverse for linear lists) *We have:*

```
$\{\mathsf{linear_{next}.p}\}$
$\mathsf{next, s :=reverse.next.p}$
```

$\sqsubseteq$

```
$\mathsf{next,\ s,\ r:=next[p \leftarrow nil],\ p,\ next.p}$
$\mathsf{while\ r \neq nil\ do}$
    $\mathsf{next,\ s,\ r:=next[r \leftarrow s],\ r,\ next.r}$
```

Lemma 36 (Refinement of reverse for loop lists) *We have:*

```
    $\{\mathsf{loop}_{\mathsf{next}}.\mathsf{p}\}$
    $\mathsf{next}, \mathsf{s} := \mathsf{reverse.next.p}$

$\sqsubseteq$

    $\mathsf{next},\ \mathsf{s},\ \mathsf{r} := \mathsf{next}[\mathsf{p} \leftarrow \mathsf{nil}],\ \mathsf{p},\ \mathsf{next.p}$
    while $\mathsf{r} \neq \mathsf{nil}$ do
        $\mathsf{next},\ \mathsf{s},\ \mathsf{r} := \mathsf{next}[\mathsf{r} \leftarrow \mathsf{s}],\ \mathsf{r},\ \mathsf{next.r}$
```

Proof.

```
    $\{\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{loop}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{q} = \mathsf{last}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{h} = \mathsf{next}_0.\mathsf{q}\}$
    $\mathsf{next},\ \mathsf{s} := \mathsf{reverse.next}_0.\mathsf{p}$

  $=$ {Definition 32 and multiple assignment}

    $\{\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{loop}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{q} = \mathsf{last}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{h} = \mathsf{next}_0.\mathsf{q}\}$
    $\mathsf{next},\ \mathsf{s} := \mathsf{preverse.next}_0.\mathsf{h.q.q},\ \mathsf{p}$

  $=$ {Lemma 33}

    $\{\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{loop}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{q} = \mathsf{last}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{h} = \mathsf{next}_0.\mathsf{q}\}$
    $\mathsf{next},\ \mathsf{s} := \mathsf{preverse}.(\mathsf{preverse.next}_0.\mathsf{p.q.nil}).\mathsf{h.p.q},\ \mathsf{p}$

  $=$ {assignment merge}

    $\{\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{loop}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{q} = \mathsf{last}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{h} = \mathsf{next}_0.\mathsf{q}\}$
    $\mathsf{next},\ \mathsf{s} := \mathsf{preverse.next}_0.\mathsf{p.q.nil},\ \mathsf{q}$
    $\mathsf{next},\ \mathsf{s} := \mathsf{preverse.next.h.p.q},\ \mathsf{p}$

  $\sqsubseteq$ {assertion introduction and assertion refinement by Lemma 33}

    $\{\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{loop}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{q} = \mathsf{last}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{h} = \mathsf{next}_0.\mathsf{q}\}$
    $\mathsf{next},\ \mathsf{s} := \mathsf{preverse.next}_0.\mathsf{p.q.nil},\ \mathsf{q}$
    $\{\mathsf{linear}_{\mathsf{next}}.\mathsf{h} \wedge \mathsf{p} = \mathsf{last}_{\mathsf{next}}.\mathsf{h}\}$
    $\mathsf{next},\ \mathsf{s} := \mathsf{preverse.next.h.p.q},\ \mathsf{p}$

  $\sqsubseteq$ {Lemma 34}

    $\{\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{loop}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{q} = \mathsf{last}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{h} = \mathsf{next}_0.\mathsf{q}\}$
    $\mathsf{next},\ \mathsf{s} := \mathsf{preverse.next}_0.\mathsf{p.q.nil},\ \mathsf{q}$
    $\mathsf{next},\ \mathsf{s},\ \mathsf{r} := \mathsf{next}[\mathsf{h} \leftarrow \mathsf{q}],\ \mathsf{h},\ \mathsf{next.h}$
    while $\mathsf{r} \neq \mathsf{nil}$ do
        $\mathsf{next},\ \mathsf{s},\ \mathsf{r} := \mathsf{next}[\mathsf{r} \leftarrow \mathsf{s}],\ \mathsf{r},\ \mathsf{next.r}$

  $\sqsubseteq$ {Lemma 34 with $\alpha = \mathsf{loop}_{\mathsf{next}_0}.\mathsf{p} \wedge \mathsf{h} = \mathsf{next}_0.\mathsf{q}$}
```

```
next, s, r := next[p ← nil], p, next.p
{$q \in [p]_{next_0} \wedge s \in [p : q]_{next_0} \wedge r = next_0.s \wedge \alpha$}
while $s \neq q$ do
    next, s, r := next[r ← s], r, next.r
    {$q \in [p]_{next_0} \wedge s \in [p : q]_{next_0} \wedge r = next_0.s \wedge \alpha$}
{$q \in [p]_{next_0} \wedge s = q \wedge r = next_0.s \wedge \alpha$}
next, s, r := next[h ← q], h, next.h
while $r \neq nil$ do
    next, s, r := next[r ← s], r, next.r
```

$\sqsubseteq$ {assertion refinement}

- $q \in [p]_{next_0} \wedge s \in [p : q]_{next_0} \wedge \alpha \ \Rightarrow next_0.s \neq nil$

```
next, s, r := next[p ← nil], p, next.p
{$r \neq nil$}
while $s \neq q$ do
    next, s, r := next[r ← s], r, next.r
    {$r \neq nil$}
{$s = q \wedge r = h \wedge r \neq nil$}
next, s, r := next[h ← q], h, next.h
while $r \neq nil$ do
    next, s, r := next[r ← s], r, next.r
```

$\sqsubseteq$ {using assertion – while and introducing if statement}

```
next, s, r := next[p ← nil], p, next.p
while $s \neq q \wedge r \neq nil$ do
    next, s, r := next[r ← s], r, next.r
if $r \neq nil$ then
    next, s, r := next[r ← s], r, next.r
    while $r \neq nil$ do
        next, s, r := next[r ← s], r, next.r
```

$\sqsubseteq$ {unfolding while}

```
next, s, r := next[p ← nil], p, next.p
while $s \neq q \wedge r \neq nil$ do
    next, s, r := next[r ← s], r, next.r
while $r \neq nil$ do
    next, s, r := next[r ← s], r, next.r
```

$\sqsubseteq$ {merge while}

```
$\mathsf{next, s, r := next[p \leftarrow nil], p, next.p}$
$\mathsf{while\ r \neq nil\ do}$
    $\mathsf{next, s, r := next[r \leftarrow s], r, next.r}$
```

We see that in both cases linear and loop lists we obtained the same algorithm.

Theorem 37 (Loop checker) *The following Hoare total correctness triple is true*

$$\begin{array}{ll}
\mathsf{next} = \mathsf{next}_0 \wedge \mathsf{p} = \mathsf{p}_0 \wedge \mathsf{list}_{\mathsf{next}}.\mathsf{p} \wedge 1 < |\mathsf{p}|_{\mathsf{next}} & \\
\{\!| & \\
\quad \mathsf{next},\ \mathsf{s} := \mathsf{reverse}.(\mathsf{next},\mathsf{p}) & \\
\quad \mathsf{hasloop} := (\mathsf{s} = \mathsf{p}) & (3)\\
\quad \mathsf{next},\ \mathsf{p} := \mathsf{reverse}.(\mathsf{next},\mathsf{s}) & \\
|\!\} & \\
(\mathsf{next},\mathsf{p}) = (\mathsf{next}_0,\mathsf{p}_0) \wedge \mathsf{hasloop} = \mathsf{loop}_{\mathsf{next}_0}.\mathsf{p}_0 &
\end{array}$$

Proof. Using the Hoare [14] assignment and sequential composition rules and then the Theorems 28 and 32.

If we replace in 3 the specification statements $\mathsf{next},\ \mathsf{s} := \mathsf{reverse}.(\mathsf{next},\mathsf{p})$ and $\mathsf{next},\ \mathsf{p} := \mathsf{reverse}.(\mathsf{next},\mathsf{s})$ with their refinements then we get a Hoare total correctness triple for a program that tests if a list has a loop. The program computes the result in linear time, does not need additional memory, and leaves the list unchanged.

6 Conclusions

We have developed a model for pointer programs suitable for refinement calculus. Our model is based on representing the fields of pointer structures as global functions from pointers to values. All pointer operations (allocating a new pointer, disposing, accessing a field, and updating a field) become simple assignments. The model is specialized for single linked lists. We have also introduced an induction theorem and an induction definition principle for lists.

Using the model for lists we have refined a specification for checking whether a list has a loop by reversing the list into an executable program. The specification was given by an inductive definition on lists, and we have gotten the properties that the loop invariant should satisfy for free. Many works on pointers [23, 17, 6, 5] have treated the example of reversing a single linked list, but none of these treats the loop list case. The real complexity comes in when one tries to reverse a loop list because although the same algorithm can be used, part of the list is traversed twice.

We have also implemented the theory for single linked lists in PVS [20] and proved some of the mentioned properties. This shows that a complete mechanization of our theory is possible.

In future work we intend to investigate if the approach is practical for other pointer structures. For example, the induction theorem can, in principle, be used to prove properties about any kind of pointer structure; however, the induction definition principle, which has simplified the reasoning about linked lists, can be used only on tree like structures.

References

[1] R. Back and V. Preoteasa. Reasoning about recursive procedures with parameters. Technical Report 500, TUCS - Turku Centre for Computer Science, January 2003.

[2] R. Back and J. von Wright. *Refinement Calculus. A systematic Introduction.* Springer, 1998.

[3] R. Back and J. von Wright. Reasoning algebraically about loops. *Acta Inform.*, 36(4):295–334, 1999.

[4] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, 1982.

[5] R. Bird. Functional pearl: Unfolding pointer algorithms. *Journal of Functional Programming*, 11(3):347–358, May 2001.

[6] R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction.* Springer-Verlag, 2000.

[7] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[8] M. Butler. Calculational derivation of pointer algorithms from tree operations. *Science of Computer Programming*, 33(3):221–260, 1999.

[9] C. Calcagno, S. Ishtiaq, and P. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *ACM-SIGPLAN 2nd International Conference on Principles and practice of Declarative Programming (PPDP 2000).* ACM Press, September 2000.

[10] A. Church. A formulation of the simple theory of types. *J. Symbolic logic*, 5:56–68, 1940.

[11] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.

[12] R. Floyd. Assigning meanings to programs. In *Proc. Sympos. Appl. Math., Vol. XIX*, pages 19–32. Amer. Math. Soc., Providence, R.I., 1967.

[13] M. Gordon and T. Melham, editors. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993. A theorem proving environment for higher order logic, Appendix B by R. J. Boulton.

[14] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[15] S. Ishtiaq and P. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–26. ACM Press, 2001.

[16] D. Luckham and N. Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):226–244, 1979.

[17] B. Meyer. Towards practical proofs of class correctness. In *ZB 2003: Formal Specification and Development in Z and B, Lecture Notes in Computer Science*, volume 2651, pages 359–387. Springer-Verlag, 2003.

[18] J. M. Morris. A general axiom of assignment, assignment and linked data stuctures, a proof of the Schorr-Waite algorithm. In *Theoretical Foundations of Programming Methodology, Lecture notes of an International Summer School*, pages 25–51. D. Reidel Publishing Company, 1982.

[19] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer science logic (Paris, 2001)*, volume 2142 of *Lecture Notes in Comput. Sci.*, pages 1–19. Springer, Berlin, 2001.

[20] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Clavert. *PVS Language Reference*, 1999.

[21] R. Paige, J. Ostroff, and P. Brooke. Formalising Eiffel reference and expanded types in PVS. In *Proc. International Workshop on Aliasing, Confinement, and Ownership in Object-Oriented Programming, Darmstadt, Germany*, July 2003.

[22] J. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*, 2000.

[23] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science.* IEEE, July 2002.