Master Thesis

Formal Verification of a C-Library for Strings

Artem Starostin
starostin@wjpserver.cs.uni-sb.de

Saarland University, Computer Science Department
Institute for Computer Architecture and Parallel Computing
Prof. Dr. W. J. Paul

March 2006

Eidesstattliche Erklärung

Hiermit erkläre ich, Artem Starostin, an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Ich habe diese Arbeit keinem anderen Prüfungsamt vorgelegt.

Saarbrücken, den 27. März 2006

Acknowledgements

I would like to express my gratitude to Alexandra Tsyban for all her help.

Contents

List of Figures

Chapter 1

Introduction

It is generally acknowledged that computer systems nowadays are universal and omnipresent. They are crucial to the functionality of systems, for example, in automotive engineering, in electronic banking, in medical technology. Our confidence in computer systems makes their reliability of large social importance. The absence of irritating and dangerous errors in them becomes one of the major factors.

On the one hand soft- and hardware failures can be not hazardous to our lives but may have significant consequences for the manufacturers. The dramatically known bug in Intel Pentium floating-point division unit caused a loss of 475 million US dollar to replace faulty microprocessors. On the other hand defects in computer systems can be really catastrophic. The cause of the Ariane 5 rocket[1] crash in 1996 was traced to a bug in the control software that calculated the rocket's position. Despite rigorous testing of the software, the problem went unnoticed.

It is not trivial to ensure oneself that a computer system does what it is intended to do. This task becomes even harder today when the complexity of computer systems grows rapidly. Testing with various inputs does not suffice because no one, in general, knows all the inputs that might cause a system to fail. Running a system on all possible inputs will consume too much time. An alternative approach is the *formal verification*. It exploits mathematical and logical methods to prove that a system is correct. Pioneered by Dijkstra, Floyd, Hoare, and others verification becomes the most convincing method to establish absence of errors in soft- and hardware.

Verisoft[2] is a long-term research project whose main goal is the formal verification of a complete computer system. The computer system — called the Academic system — basically consists of hardware, an operating system and user applications. User programs are written in a high level programming language, a compiler for which is being verified in the frame of the

[1] http://www.arianespace.com
[2] http://www.verisoft.de

project. Together with a compiler several libraries which implement basic data structures and algorithms on them are provided. This diploma thesis is aimed at the formal verification of a library for strings.

In Verisoft project computer aided verification is being performed in order to prevent human errors conducted by the scientists involved. A variety of tools is used to establish the correctness of different layers of a computer system. This work involves the usage of a general interactive theorem prover Isabelle/HOL. The formal proof of the library was made in a verification environment built in this theorem prover. Since the machine readable proofs and specifications do not fit for the paper-and-pencil presentation on the pages of this thesis, we introduce mathematical notations which allow us to omit some details.

Outline

The rest of this thesis is organized in six chapters.

- In Chapter 2 we point out the goals of this work and the strategy we use to achieve them. We continue with a brief overview of the C0 programming language which we use to implement the target library. This chapter also covers the basic notations which appear in the thesis.
- In Chapter 3 we define the notion of a formal program specification with respect to the Hoare Logic, a formal system which we use for the verification of the library. We present a set of sound rules for the establishment of C0 program correctness and conclude with an example of the application of these rules.
- In Chapter 4 we formalize the idea of references and pointer structures in a way it is done in the computer aided verification environment. We present abstract relations between implementation and specification of pointer primitives which play role of building blocks for the string data structure.
- In Chapter 5 we present an abstraction mapping for strings. We comment details of the library implementation and specify the functions formally. We give sufficient invariants and ranking functions for each loop of these functions as well.
- In Chapter 6 we formally prove the relatively complex function from the library to give a taste of steps and techniques we used for formal verification with the help of computer aided tools.
- Chapter 7 concludes with a summary of the work.

Chapter 2

Basics

We begin this chapter by setting the goals of this work and sketching the strategy we use to achieve them. We continue with a brief overview of the C0 programming language which is used for the implementation of the target library. The chapter is concluded with the basic notations which we use in the whole thesis.

2.1 Goals and the Strategy

The goal of this work is to continue the work started in [Pre05] in order to obtain a formally verified C-library for strings. By *formal verification* we understand the act of proving or disproving the correctness of a system *implementation* with respect to a certain *specification* using *formal methods*. Formal methods refer to mathematically based techniques for the specification, development and verification of computer systems. Since the complexity of computer systems is relatively high, their formal verification is feasible with the help of *formal tools* — verification environments, model checkers, interactive theorem provers, etc.

We implement the target library in the C0 programming language (see section 2.2 for details).

The specification counterpart is formulated in the Verification environment for sequential imperative programs [Sch05] in Isabelle/HOL [NPW04]. Since this environment exploits Hoare Logic [Hoa69] we will refer to it as *Hoare Logic environment*. Isabelle/HOL is a general interactive theorem prover in higher-order logic. The Hoare Logic environment and Isabelle/HOL are the formal tools we use.

To reason about the implementation correctness with respect to the specification we automatically translate the C0 code into the internal language of Hoare Logic environment. After precise analysis of the specification and implementation we annotate each loop statement of this translation with a sufficient invariant. Such annotated implementation together with the

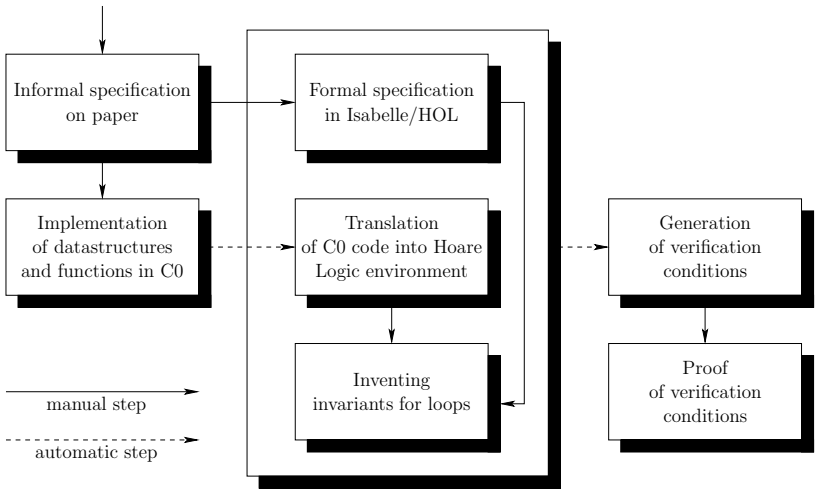

Figure 2.1: Milestones of obtaining formally verified software

specification allows to generate verification conditions (see section 3.2.2) in the Hoare logic environment. The interactive proof of these conditions in Isabelle/HOL finishes the verification process.

The overall strategy is depicted in figure 2.1.

2.2 The C0 Programming Language

Programming languages, by their own nature, are quickly created and changed. On the one hand, their evolution brings the variety of features which ease the software development process. On the other hand, these features complicate the axiomatization of programming language constructs (see chapter 3), or even make it impossible [Cla79]. Therefore the formal reasoning about the programs written in such languages is unfeasible. Oppositely to the simplicity of program development, program verification requires the simplicity of a programming language. The verification issues caused the design of the *C0 programming language* in the frame of the Verisoft project.

2.2.1 General Restrictions

The C0 programming language is a restricted version of ANSI C [C99]. In this thesis we only sketch the significant restrictions, the type system, and the abstract syntax of C0. Additional information about the C0 programming language can be found in [LPP05].

The major restrictions are:

- no initialization during declarations, except for a constant declaration;

- no side-effects inside expressions;
- no function calls inside expressions;
- the size of arrays is fixed at the compile time;
- no variable declarations in functions after the first statement;
- only one return statement which is the last control command in each function;
- no pointer arithmetic;
- no pointers to local variables;
- no pointers to functions;
- no *void* pointers, i.e. all pointers are typed.

2.2.2 Type System

The built-in type system of C0 is limited to the four basic types, namely:

- 32-bit signed integers: `int` $= \{-2^{31}, \ldots, 2^{31} - 1\}$;
- 32-bit unsigned integers: `unsigned int` $= \{0, \ldots, 2^{32} - 1\}$;
- Boolean: `bool` $= \{$`true`, `false`$\}$;
- 8-bit signed integers: `char` $= \{-128, \ldots, 127\}$.

Based on these simple types one can construct complex types. They are:

- Typed pointers: `typ *x;`
- Arrays: `typ a[size];`
- Structures: `struct styp {typ data};`

Here `typ` stands for the basic or complex type, `styp` — for the newly declared structural type, and `size` — for a constant of integer type.

2.2.3 Statements

The statements allowed in C0 are:

- **Assignment**
  `l = expr;`
  The assignment is possible between expressions of basic types, structures, or pointers. The types of left-side expression `l` and right-side expression `expr` must be the same. During the execution of an assignment statement expression `expr` is evaluated and the value of `l` is changed to the value of `expr`.

- **Loop**
  `while (cond) { stmts }`
  The test-condition `cond` must be of type `bool`. The execution of a loop statement proceeds as follows. First condition `cond` is evaluated. In case `cond` = `false` the execution is complete. Otherwise, statements `stmts` are executed and `cond` is tested again. The action is repeated until the value of `cond` becomes `false`.

- **Conditional**
  `if (cond) { stmts }`
  `if (cond) { stmts_1 } else { stmts_2 }`
  The test-condition `cond` must be of type `bool`. During the execution of a conditional statement condition `cond` is firstly evaluated. In case `cond` = `true` the program execution continues with statements `stmts` (with `stmts_1` for the second variant of conditional). In case `cond` = `false` the execution of conditional is finished (continues with statements `stmts_2` for the second variant).

- **Function call**
  `l = foo(expr_1, ..., expr_n);`
  The type of variable `l` and the types of all parameters must respect the declared signature of the function `foo`. The function call is executed in the following way. First actual parameters `expr_1, ..., expr_n` are evaluated, then these values are assigned to the respective formal parameters of `foo`. The function body is executed and the return-result is assigned to variable `l`.

- **Return**
  `return expr;`
  The return statement is used in functions to yield values. The type of expression `expr` must respect the signature of the function where it is used.

- **Allocation of dynamic memory**
  `l = new(typ);`
  The type of variable `l` must be `*typ`. The desired behavior of this statement is to allocate a block of memory sufficient for the size of type `typ` during the run-time of a program (see section 4.1 for the description of dynamic memory mechanisms).

2.3 Notation

In this section we introduce some useful notations for the types which we will use in the specifications, an universal container (list), and review some definitions from the field of logic which we will exploit later.

For the whole thesis we define the following data types:

$$\begin{aligned}
\mathbb{B} &\stackrel{\text{def}}{=} \{\mathsf{True}, \mathsf{False}\},\\
\mathbb{N} &\stackrel{\text{def}}{=} \text{set of all natural numbers } \textit{including } 0,\\
\mathbb{Z} &\stackrel{\text{def}}{=} \text{set of all integers},\\
\mathbb{N}_{\mathrm{n}} &\stackrel{\text{def}}{=} \{x \mid x \in \mathbb{N} \wedge x < n\},\\
\mathbb{C}\mathrm{har} &\stackrel{\text{def}}{=} \mathbb{N}_{256}.
\end{aligned}$$

Thus the set $\mathbb{B}$ corresponds to the type-domain of two-valued Boolean algebra. $\mathbb{N}$ and $\mathbb{Z}$ are the notations for natural numbers and integers. Note that the set of natural numbers in our notation includes zero. Therefore when we say that a number is *natural* we suppose that it could be zero. $\mathbb{N}_{\mathrm{n}}$ is the notation for a set of n consecutive natural numbers starting with zero. Finally $\mathbb{C}\mathrm{har}$ is a special data type which is isomorphic to the type `char` in C0. It will be basically used to represent characters in abstract mathematical notation[1].

In the specifications of programs we often need to claim the certain properties of the sequences of elements. For this purpose we introduce an universal container — an abstract list.

Definition 2.1 (Abstract List) *An **abstract list** of length $n \in \mathbb{N}$ whose elements are of type $\mathbb{T}$ is a mapping*

$$l : \mathbb{N}_{\mathrm{n}} \to \mathbb{T}.$$

An abstract list is uniquely defined by an n-tuple of values $(l[0], l[1], \ldots, l[n-1])$. As a shorthand for this tuple we use $l[0 : n-1]$. We denote the set of all abstract lists of length n of type $\mathbb{T}$ by

$$L^n_{\mathbb{T}} \stackrel{\text{def}}{=} \{l \mid l : \mathbb{N}_{\mathrm{n}} \to \mathbb{T}\}.$$

We denote the empty list for all types by the polymorphic constant

$$[\,] \stackrel{\text{def}}{\in} L^0_{\mathbb{T}}.$$

We denote the set of all abstract lists of type $\mathbb{T}$ by

$$L^*_{\mathbb{T}} \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} L^n_{\mathbb{T}}.$$

Note that when it is clear from the context that we speak about an abstract list we omit the adjective "abstract".

[1] Precisely speaking it represents ASCII codes of characters.

Definition 2.2 (Abstract List Concatenation) *Let* $l[0 : n-1]$ *and* $k[0 : m-1]$ *be abstract lists whose elements are of type* $\mathbb{T}$. *A* ***concatenation*** *is a function*

$$\odot : L_{\mathbb{T}}^{n} \times L_{\mathbb{T}}^{m} \to L_{\mathbb{T}}^{n+m},$$

such that

$$l[0 : n-1] \odot k[0 : m-1] \stackrel{\text{def}}{=} (l[0], l[1], \ldots, l[n-1], k[0], k[1], \ldots, k[m-1]).$$

Suppose one wants to concatenate n *abstract lists* $a_0, a_1, \ldots, a_{n-1}$, *where* $a_i \in L_{\mathbb{T}}^{*}$ *for* $0 \leq i < n$. *For this purpose we will write:*

$$\bigodot_{0 \leq i < n} a_i \stackrel{\text{def}}{=} a_0 \odot a_1 \odot \ldots \odot a_{n-1}.$$

Note that the abstract list concatenation definition is implicitly quantified over all n and m. We do not introduce a new operator for the concatenation of a single element to a list. In such cases we implicitly transform this element to a list of length one containing this element.

In the following definition we introduce three useful operations on abstract lists. They are the *set* operation which returns the set of list elements, the *reverse* operation which yields the list whose elements order is reversed, and the *length* operation.

Definition 2.3 (Operations on Abstract Lists) *The* ***set*** *operation is a function*

$$\mathcal{S} : L_{\mathbb{T}}^{*} \to 2^{\mathbb{T}}$$

which yields for an abstract list $l \in L_{\mathbb{T}}^{*}$ *the set of its elements:*

$$\mathcal{S}(l[0 : n-1]) \stackrel{\text{def}}{=} \{l[i] \mid 0 \leq i < n\}.$$

The ***reverse*** *operation is a function*

$$^{-1} : L_{\mathbb{T}}^{*} \to L_{\mathbb{T}}^{*}$$

which yields for an abstract list $l \in L_{\mathbb{T}}^{*}$ *an abstract list obtained by reversing its elements:*

$$l^{-1}[0 : n-1] \stackrel{\text{def}}{=} (l[n-1], l[n-2], \ldots, l[0]).$$

The ***length*** *operation is a function*

$$|\ | : L_{\mathbb{T}}^{*} \to \mathbb{N}$$

which yields for an abstract list $l \in L_{\mathbb{T}}^{*}$ *its length, i.e. the number of elements it contains:*

$$|l[0:n-1]| \stackrel{\text{def}}{=} n.$$

Let us illustrate these notions with an example. Suppose an abstract list $l \in L^*_{\mathbb{N}}$ such that $l = (3, 1, 0, 1)$. The following equalities then hold:

$$\begin{aligned} \mathcal{S}(l) &= \{0, 1, 3\}, \\ l^{-1} &= (1, 0, 1, 3), \\ |l| &= 4. \end{aligned}$$

The definitions below (2.4-2.8) we will use for the presentation of a formal system for reasoning about program correctness (see section 3.2.1) and for formal modeling of pointer programs (see chapter 4) with their subsequent verification (refer to chapter 6).

Definition 2.4 (Substitution) *The formula $F[x := e]$ is a **substitution** instance of the formula F obtained by replacing all free occurrences of x in F by e.*

Consider an example. Assume a well-typed formula: $F = x \longrightarrow \exists x.x \wedge y$. The result of substitution $F[x := z \vee x]$ is $z \vee x \longrightarrow \exists\, x\,.\, x \wedge y$.

Note that several substitutions we will separate by comas, e.g.

$$F[x := e_1, y := e_2, z := e_3] = ((F[x := e_1])[y := e_2])[z := e_3].$$

The problem of logical operators precedence is fixed by the following convention:

Convention 2.5 (Logical Operators Precedence) *The precedence order of logical operators is: $\wedge, \vee, \longrightarrow$, i.e. conjunction binds stronger than disjunction which binds stronger than implication.*

In particular $x \vee y \wedge z \longrightarrow u \vee w$ is the same as $((x \vee (y \wedge z)) \longrightarrow (u \vee w))$. Also note that all the arithmetic operators bind stronger than logical.

Definition 2.6 (Hilbert's Choice Operator) *Let $\mathbb{T}$ be a set. **Hilbert's choice operator** is a function*

$$\varepsilon : 2^{\mathbb{T}} \to \mathbb{T},$$

such that for every subset $t \subseteq \mathbb{T}$, $\varepsilon(t)$ yields an element of t. Formally:

$$\forall\, t \in 2^{\mathbb{T}}\,.\,(\exists x \in \mathbb{T}\,.\, x \in t) \longrightarrow \varepsilon(t) \in t.$$

Assume a set of natural numbers $t = \{1, 4, 6, 9\}$. Then the result of the operation $\varepsilon(t)$ may be either 1, 4, 6, or 9.

Definition 2.7 (Function Self Composition) *Let f be a function. Then f^i for $i \in \mathbb{N}$ denotes an i times **self composition** of the function f:*

$$f^i(x) = \begin{cases} x & \text{if } i = 0 \\ f^{i-1}(f(x)) & \text{otherwise} \end{cases}.$$

Definition 2.8 (Function Update) *Let $f : \mathbb{T}_1 \to \mathbb{T}_2$ be a function, and $x \in \mathbb{T}_1$, $y \in \mathbb{T}_2$. We write $g = f[x \to y]$ to introduce the new function $g : \mathbb{T}_1 \to \mathbb{T}_2$, such that for $z \in \mathbb{T}_1$:*

$$g(z) = \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases}.$$

Chapter 3

Hoare Logic

This chapter gives a notion of a formal program specification according to Hoare Logic [Hoa69], a formal system which we use for the verification of the library. The rules for the establishment of C0 program correctness are covered and an example of their application is presented.

3.1 Program Specification

One of the most common approaches to program specification and verification is the usage of *Hoare Logic*. Hoare Logic is a formal system developed by the British computer scientist C.A.R. Hoare and subsequently refined by Hoare and other researchers. In [Hoa69, Hoa71] Hoare proposes an idea that all the properties of a program and all the consequences of its execution can, in general, be found out from the text of the program itself by the purely deductive reasoning. *Deduction* means the application of sound inference rules to sets of valid axioms. Since in most cases program specifications play the role of axioms, i.e. the set of equations we trust in, we start with a formalization of the idea of specification.

3.1.1 Pre- and Postconditions

A program specification is the definition of what a computer program is expected to do. A *formal* program specification has a certain meaning defined in mathematical terms. As it follows from section 2.1 we use the language of higher-order logic in the Hoare Logic environment for the formal specification of programs. For the paper presentation of the specification we formulate the equivalent specifications using abstract mathematical language. We continue with the intention of a formal program specification.

In general, computer programs can be interpreted as input/output relations over a set of configurations. We call the configurations of a program its *state space*.

Definition 3.1 (State Space) *A **state space** Σ of a particular program is the collection of all:*

- *variables of basic types,*
- *flattened structure fields (i.e. every component of a structure is represented by a separate item),*
- *the signatures of all functions*

of this program together with their types.

Let us illustrate this notion with an example. Consider the following C0 program:

Listing 3.1: Example of a C0 program involving structural types

```
1  struct Container {
2    int  info;
3    bool flag;
4  };
5
6  int              x, y;
7  struct Container c;
8
9  int add(int x, int y) {
10   return x + y;
11 }
12
13 int main() {
14   c.info = x + y;
15   return 0;
16 }
```

The respective state space of this program is:

$$\begin{aligned}\Sigma = (x &: \mathbb{Z},\\ y &: \mathbb{Z},\\ c\_info &: \mathbb{Z},\\ c\_flag &: \mathbb{B},\\ add\_x &: \mathbb{Z},\\ add\_y &: \mathbb{Z},\\ res\_add &: \mathbb{Z},\\ res\_main &: \mathbb{Z}).\end{aligned}$$

Variables `x` and `y` of basic types are straightforwardly correspond to the components x and y of Σ. Variable `c` of structural type (`Container`) is represented by the components c_info and c_flag which directly correspond to the fields of the structure `Container`. Parameters of the function `add` are

represented by the components add_x and add_y of Σ. We concatenate the respective function names to the names of formal parameters to distinguish them from the existing components x and y in the state space. Finally we represent the return-results of functions `main` and `add` by res_main and res_add, respectively. Note that we will drop suffixes which correspond to the names of functions, e.g. $\_main$, $\_add$, in case we treat only one function in the context.

Note that we represent basic integer C0 types, i.e. `int` and `unsigned int` in the state space as abstract unbounded types $\mathbb{Z}$ and $\mathbb{N}$. This follows from the formal model in the Hoare Logic environment where the subtypes are implemented by means of the *guards*. Although in the frame of this work we do not use guards information about them can be found in [Sch05]. As for C0 type `char` it is represented by $\mathbb{C}$har (recall section 2.3).

It follows that programs describe binary relations over Σ. Therefore one can consider the variety of all possible programs over Σ as the powerset $2^{\Sigma\times\Sigma}$.

One of the most important properties of programs is whether they behave as we suppose them to. The intended behavior of a program can be specified by making general assertions about the state space of a program after its execution. These assertions usually do not claim certain values of each variable, but specify general properties of the values and the relationship between them. In many cases, the validity of a program result depends on the state space before the program execution as well. The properties of the initial configuration can be specified by the same type of general assertions.

Definition 3.2 (Assertion) *Let Σ be the state space. An **assertion** is a predicate drawn from the set*

$$\mathcal{A}_\Sigma \stackrel{\text{def}}{=} \{\alpha \mid \alpha : \Sigma \to \mathbb{B}\}.$$

An assertion which describes the initial values of the variables is called a *precondition*. An assertion describing the result of a program execution is called a *postcondition*. We refer to pre- and postcondition together as the formal *specification*.

3.1.2 Hoare Triple

To state the required connection between a program and its specification we introduce new notation.

Definition 3.3 (Hoare Triple) *Let $A, B \in \mathcal{A}_\Sigma$ be pre- and postconditions of a program $P \in 2^{\Sigma\times\Sigma}$, where Σ is a state space of P. A **Hoare triple** $\{A\}\, P\, \{B\}$ is a function:*

$$\mathcal{A}_\Sigma \times 2^{\Sigma\times\Sigma} \times \mathcal{A}_\Sigma \to \mathbb{B},$$

such that whenever validity of A in the initial state of P implies validity of B in the terminating state of P, $\{A\}\ P\ \{B\}$ yields True. *More precisely, there are two interpretations:*

- *a triple $\{A\}\ P\ \{B\}$ is* True *in the sense of* ***partial correctness*** *if every terminating computation of P that starts in a state satisfying A terminates in a state satisfying B;*
- *a triple $\{A\}\ P\ \{B\}$ is* True *in the sense of* ***total correctness*** *if every computation of P that starts in a state satisfying A terminates, and its final state satisfies B.*

Note that idea of the Hoare triple is also applicable to parts of programs.

We elaborate on the program correctness proofs with respect to these two criteria in the next section. First we give the proof system which allows to reason about the functional counterpart of programs. This corresponds to their partial correctness. Then we introduce the notion of loop *ranking functions*, which allows us to reason about program termination, and therefore the total correctness of programs. In addition to that, we give an idea of the *absence of run-time errors*, basically, the errors caused by the overflows and $Null$-pointers dereferencing. We show how the specifications should be extended in order to prove that the errors of this nature do not occur in a program.

3.2 Program Verification

The deductive reasoning about the correctness of a program implementation with respect to the specification requires a proof system. By the proof system we understand a set of sound rules which underline the reasoning about computer programs. In this section we present such rules. The choice of them depends on the choice of the programming language. Since we implement the target library in the C0 programming language the inference rules must respect syntax and semantics of C0. These rules were developed and embedded in the Hoare Logic environment by Schirmer (refer to [Sch05]). In this thesis we present their lite versions.

3.2.1 Proof System

Before we introduce inference rules we give the idea of *inference*.

Definition 3.4 (Inference Rule) *Given a set $\{A_1, A_2, \ldots, A_n\}$ of valid* ***premises*** *and a* ***conclusion*** *B, an object*

$$\frac{A_1\ A_2\ \ldots\ A_n}{B},$$

denotes that whenever in the course of some logical derivation the given premises have been obtained, the conclusion can be taken for granted as well. We call such an object an ***inference rule****.*

Note, that in principle the set of premises can be empty. In this case we write:

$$\frac{}{B}\,.$$

We introduce the proof system by induction on the program syntax. Each of the rules claim a relation between pre- and postconditions of a particular C0 statement and this statement itself. Therefore in all rules we assume the straightforward correspondence between the variables from the state space whose names we write in *italic* and variables from the program whose names we write in `fixed-pitch` font. The former will occur in assertions, the latter — in statements of C0.

We will refer to the rules below as to *Hoare rules*.

Assignment

Consider the assignment statement:

```
x := expr
```

Any assertion $B = f(x)$ which evaluates to True with the value of x after the assignment to x is made, must also hold evaluated with the value of $expr$, taken before the assignment is made. In other words, if $f(x)$ holds after the assignment, then $f(expr)$ must hold before. Formally, we write this exploiting the notion of substitution (definition 2.4):

Rule 3.5 (Assignment) *For an assertion B, a variable x, and an expression $expr$:*

$$\frac{}{\{B[x := expr]\}\ \texttt{x := expr}\ \{B\}}\,.$$

Sequential Composition

Let us take the sequential composition of the parts of a program:

```
stmts_1; stmts_2
```

Assume that the precondition A of `stmts_1` holds. If the proved result B' of `stmts_1` is identical with the precondition under which statements `stmts_2` yield their intended result B, then the composition `stmts_1; stmts_2` produces the intended result. Therefore:

Rule 3.6 (Composition) *For assertions A, B, B', and parts of a program* `stmts_1` *and* `stmts_2`*:*

$$\frac{\{A\}\ \texttt{stmts\_1}\ \{B'\} \quad \{B'\}\ \texttt{stmts\_2}\ \{B\}}{\{A\}\ \texttt{stmts\_1; stmts\_2}\ \{B\}}\,.$$

Loop

Consider the loop statement:

```
while (cond) do { stmts }
```

The reasoning which leads to a formulation of an inference rule for it is as follows. Suppose A is an assertion which is true on the completion of statements `stmts` execution, provided it holds in the initial state of their execution. Obviously, A is true after any number of iterations of the statements `stmts`. It is also known that the condition *cond* is False when the iteration finally terminates. Furthermore, *cond* may be assumed to be True on the initiation of `stmts`. Formally:

Rule 3.7 (Loop) *For an assertion A, a boolean expression cond, and a part of a program* `stmts`*:*

$$\frac{\{A \wedge cond\}\ \texttt{stmts}\ \{A\}}{\{A\}\ \texttt{while (cond) do \{ stmts \}}\ \{A \wedge \neg cond\}}.$$

Since A holds at any point of the loop execution we call A a *loop-invariant*.

Conditional

A Hoare triple

$$\{A\}\ \texttt{if (cond) then \{ stmts\_1 \} else \{ stmts\_2 \}}\ \{B\}$$

can hold only if the following conditions are fulfilled. First of all an execution of statements `stmts_1` and `stmts_2` must lead to a state satisfying B provided A is satisfied in the initial state. Furthermore according to the semantics of this statement *cond* must hold on the initiation of `stmts_1` and $\neg cond$ must hold before the execution of `stmts_2`. Formally:

Rule 3.8 (Conditional) *For assertions A, B, a boolean expression cond, and parts of a program* `stmts_1` *and* `stmts_2`*:*

$$\frac{\{A \wedge cond\}\ \texttt{stmts\_1}\ \{B\} \quad \{A \wedge \neg cond\}\ \texttt{stmts\_2}\ \{B\}}{\{A\}\ \texttt{if (cond) then \{ stmts\_1 \} else \{ stmts\_2 \}}\ \{B\}}.$$

Function Call

Suppose a function `foo` is implemented in a C0 program with the body `stmts` (not including the return statement):

```
res_typ foo(typ_1 x_1, ..., typ_n x_n) {stmts; return res_foo}
```

where `res_typ`, `typ_1`, ..., `typ_n` are the types of the return value and the formal parameters respectively and `res_foo` is the return value.

The idea behind the inference rule for the call statement

```
l = foo(expr_1, ..., expr_n)
```

is as follows. Let A be an assertion which holds before the execution of the procedure body `stmts`. Clearly A where all entries of formal parameters are replaced by the corresponding expressions (actual parameters) holds before the call of `foo` is made. Furthermore an execution of this call leads to a state satisfying B only in the case the execution of statements `stmts` leads to the state satisfying assertion B where all occurrences of l are replaced by the return value of `foo`. Using the substitution notation (definition 2.4), one can write formally:

Rule 3.9 (Call) *For assertions A, B, a variable l, a procedure* `foo` *such that* `stmts` *is its body, $x_1, \ldots, x_n$ are its formal parameters, res_foo is its return value, and expressions $expr_1, \ldots, expr_n$:*

$$\frac{\{A\}\ \texttt{stmts}\ \{B[l := res\_foo]\}}{\{A[x_1 := expr_1, \ldots, x_n := expr_n]\}\texttt{l := foo(expr\_1, ..., expr\_n)}\{B\}}.$$

The Rule of Consequence

The last rule we present in this thesis reflects the basics of logical reasoning. We say that assertion A is *stronger* than A' iff $A \longrightarrow A'$.

Let assertion A be stronger than A' and B' be stronger than B. If a Hoare triple $\{A'\}\,\texttt{stmts}\,\{B'\}$ is valid then we obviously can conclude that a Hoare triple $\{A\}\,\texttt{stmts}\,\{B\}$ is valid as well. Formally:

Rule 3.10 (Consequence) *For assertions A, A', B, B' and a part of a program* `stmts`*:*

$$\frac{A \longrightarrow A' \quad \{A'\}\,\texttt{stmts}\,\{B'\} \quad B' \longrightarrow B}{\{A\}\,\texttt{stmts}\,\{B\}}.$$

The soundness and completeness results of such a proof system can be found in [Apt81].

3.2.2 Weakest Precondition Strategy

Having a proof system, one needs a verification strategy to proceed. In other words one should decide how to apply rules in order to obtain a convincing correctness proof. We use the *weakest precondition strategy* for formal verification of programs.

We call *verification conditions* a mathematical lemmata which one has to prove in order to get a formal correctness proof of a program with respect to its specification. These verification conditions can be expressed by formulas in predicate logic. According to the weakest precondition strategy one takes a program and its specification and applies the Hoare rules backwards until

the program is completely eliminated. During this verification conditions are generated.

The weakest precondition strategy is based on the *weakest precondition predicate transformer* method which was developed by E.W. Dijkstra [Dij76]. The idea behind it is as follows. Suppose one has a program P and its postcondition B. The goal is to determine the precondition A', such that the triple $\{A'\}\, P\, \{B\}$ evaluates to True. In general there could be arbitrary many preconditions A' satisfying this criteria. However, there is exactly one non-redundant precondition describing the *maximal* set of possible initial states of P, such that its execution leads to a state satisfying B. Such precondition is called the *weakest precondition*.

Let us consider a program composed of n statements:

$$P = \texttt{stmt\_1; stmt\_2; ...; stmt\_n}.$$

Let A and B be its pre- and postconditions, respectively. Using the weakest precondition strategy we start with `stmt_n` and B applying the Composition rule (3.6) and the Hoare rule which syntactically matches `stmt_n`. We produce A_n and accumulate some verification conditions. A_n is the weakest precondition for the statement `stmt_n` and becomes the postcondition for the rest of the program which ends with the statement $n-1$. Then we take this statement and A_n and repeat the procedure. Finally, we end up with the sequence:

$$\{A_1\}\ \texttt{stmt\_1}\ \{A_2\}\ \texttt{stmt\_2}\ \ldots\ \{A_n\}\ \texttt{stmt\_n}\ \{B\},$$

and produce a list of lemmata. Now one has to prove the verification conditions: the generated lemmata and also $A \longrightarrow A_1$. If we succeed in this, we can be sure that P meets its specification.

Apart from the implication from a precondition to the weakest precondition verification conditions are generated only for certain statements, e.g. loops. Automatic verification of loop statements is very difficult in general [Pop03]. Therefore, all the loops in a program have to be annotated manually with *invariants*. An invariant is a formula in predicate logic that holds during every iteration of the loop and on its termination.

For instance, for a program containing exactly one non-trivial loop, the application of the weakest precondition strategy leaves three subgoals [MN04]: implication from the precondition to the invariant, preservation of the invariant, and validity of the postcondition upon loop termination.

As soon as all the loops of a program are supplemented with proper invariants it is possible to apply Hoare rules with respect to the weakest precondition strategy automatically. Efficient algorithms for this exist. In particular one of them is implemented in the Hoare Logic environment of Isabelle/HOL as the tactic *vcg*[1]. The details of its behavior can be found in [Sch05]. Note that all the proofs in this work involve usage of this tactic.

[1]VCG stands for Verification Condition Generator.

We illustrate the details of the weakest precondition strategy with an example in the end of this chapter (section 3.3).

3.2.3 Proving Termination

The two main constructs of the C0 programming language that may cause a program not to terminate are loops and recursion. Since we do not use the latter in the implementation of the library for strings we focus on the *loop* termination. The basic idea is to justify termination of a loop by a well-founded relation over the program state space.

We will specify a well-founded relation by the loop *ranking function*. This is a function from the program state space to the natural numbers, such that whenever it is applied to the two consecutive states of a loop, the yielded values strictly decrease.

Definition 3.11 (Ranking Function) *Let Σ be the state space, and let $\sigma_0, \sigma_1, \ldots$ be the sequence of the states corresponding to the iterations of a loop whose termination we have to prove. A* ***ranking function*** *for this loop is a function drawn from the set*

$$\mathcal{R}_\Sigma \stackrel{\text{def}}{=} \{\rho \mid \rho : \Sigma \to \mathbb{N} \ \wedge \ \forall i \,.\, \rho(\sigma_{i+1}) < \rho(\sigma_i)\}.$$

Thus, in order to prove the function with respect to the total correctness criterion we have to annotate its every loop with the sufficient ranking function. The verification condition generator of the Hoare Logic environment will automatically generate the subgoals asking the user to show that the ranking function evaluated over the variables of the loop's next state has the value less then being evaluated over the variables in the current state.

3.2.4 Absence of Run-Time Errors

It is the case that even the total correctness proof of a program cannot give absolute confidence that the program will not fail. The reason for this is that certain properties of the program can be checked only during the execution of this program. Typical run-time faults are array bound violation, dereferencing $Null$ pointers or arithmetical overflow.

Fortunately, the techniques introduced in [Sch05] allow us to guard expressions of a statement against run-time faults without running or simulating a program. The concept of *guards* was implemented in the Hoare Logic environment. Guards make the potential run-time fault explicit, since the expressions in the Hoare Logic environment will never fail because they are ordinary higher-order logic expressions over the variables of abstract unbounded types, such as $\mathbb{Z}$, $\mathbb{N}$, etc.

Currently guards are generated for:

- overflows and underflows of numbers,

- division by zero,
- dereferencing of $Null$ pointer, and
- array bound violations.

In order to show that none of the above faults occur, the verification condition generator analyses the program source code and produces a sufficient subgoal for every fault-potential (sub-) expression [2]. Usually, these subgoals are straightforward to prove.

For every (sub-) expression of the form `x` $\circ$ `y` with $\circ \in \{+, -, *, /\}$ the subgoal basically of the form

$$x \circ y \leq \texttt{MAX\_NAT} \text{ or}$$
$$\texttt{MIN\_INT} \leq x \circ y \leq \texttt{MAX\_INT}$$

will be generated depending on the type of the expression. The constants `MAX_NAT`, `MIN_INT`, and `MAX_INT` are defined depending on the arithmetic of the underlying machine on which a program is supposed to run. During the verification of the string library we assumed the following values:

$$\begin{aligned} \texttt{MAX\_NAT} &\stackrel{\text{def}}{=} 2^{32} - 1 \\ \texttt{MIN\_INT} &\stackrel{\text{def}}{=} -2^{31} \\ \texttt{MAX\_INT} &\stackrel{\text{def}}{=} 2^{31} - 1. \end{aligned}$$

The generated because of guards subgoals for the remaining run-time errors are even simpler. For the division by zero we demand the divisor be different from zero. For the dereferencing of $Null$ pointer error we require the pointer to be dereferenced have the non-$Null$ value. Finally, for every array index we prove that it lies within the bounds of array.

3.3 Verification Example

In this section we give a "toy" example of program verification using all previously described techniques. We prove the total correctness of a program and show the absence of run-time error in it. We consider a simple program which computes a square of a number via repeated addition. The C0 source code of this program is presented in listing 3.2. We assume variables `x`, `y`, and `z` to be of type `int`.

Listing 3.2: Computation of a square via addition

```
1 y = 0; z = 0; while (z <= x - 1) {y = y + 2*z + 1; z = z + 1}
```

[2] By the *subexpression* we understand one elementary operation in the expression according to its syntactical tree.

The program is intended to compute a square of `x` and store the result in `y` provided `x` is non-negative. Therefore the formal specification of the program is as follows:

$$\begin{aligned} A &= x \geq 0 \wedge x^2 \leq \texttt{MAX\_INT}, \\ B &= y = x^2, \end{aligned}$$

where A is a precondition and B is a postcondition [3]. The term $x^2 \leq \texttt{MAX\_INT}$ in A is the necessary assumption in order to show the absence of overflows.

Showing Partial Correctness

The overall process of Hoare rules application according to the weakest precondition strategy is depicted in figure 3.1. We denote the weakest precondition to be determined at any step by $\{?\}$. We instantiate this placeholder with a sufficient condition as soon as possible. Further I stands for the loop invariant. It has to be invented in order to show the generated verification conditions.

In this example Hoare rules application terminates yielding four terminating nodes (leaves of the proof-tree). Two of them are easily proved by the Assignment rule. Recall that it does not have any premises and hence the proof of a subgoal finishes with an application of this rule. The other two subgoals are accumulated verification conditions. We denote them by VC_2 and VC_3. According to our classification they correspond to the preservation of the invariant and to the implication from the invariant to the postcondition respectively. The remaining first verification condition is the implication from the precondition A to the weakest precondition. In our case the weakest precondition is $I[z := 0, y := 0]$.

Let us summarize. The following verification conditions have to be shown in order to establish the correctness of the program 3.2 with respect to its formal specification:

VC_1: $A \longrightarrow I[z := 0, y := 0]$,

VC_2: $I \wedge z \leq x-1 \longrightarrow I[z := z + 1, y := y + 2 * z + 1]$,

VC_3: $I \wedge z > x-1 \longrightarrow B$.

The invariant which suffices for this function is $I = y = z^2 \wedge z \leq x$. Substituting it as well as A and B in VC_1-VC_3 results in three predicate logic statements whose validity is trivial to establish.

[3] We stick to the following convention: whenever we define an assertion $A \in \mathcal{A}_\Sigma$, we write its definition in the form $A = expr$ instead of $A(\sigma) = expr$ omitting the argument $\sigma \in \Sigma$, a state in which the expression $expr$ must be evaluated. The same will hold for the ranking functions.

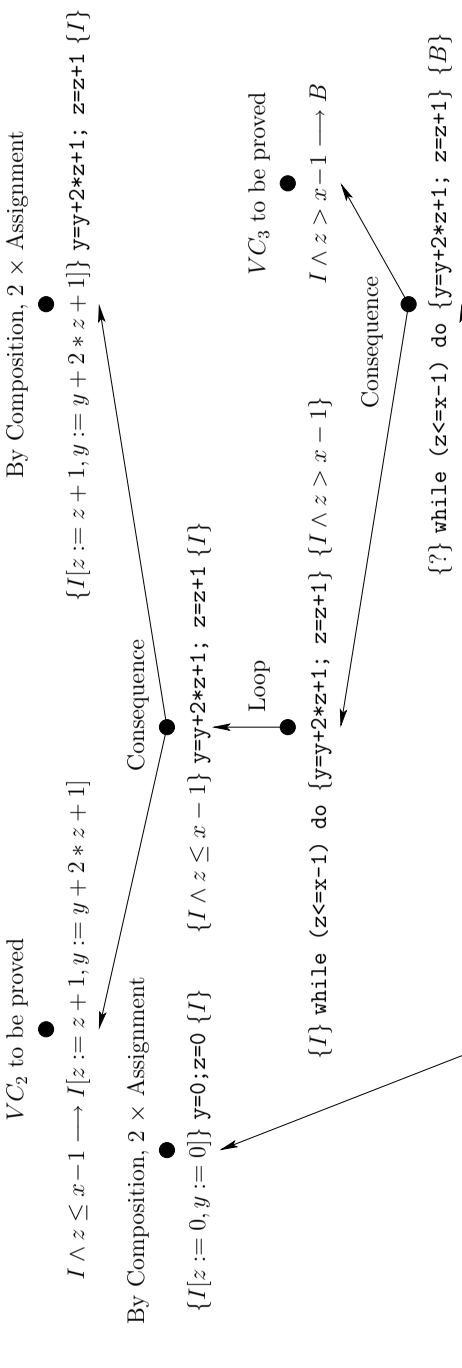

Figure 3.1: Tree of Hoare rules application

Showing Termination

In order to prove that the function's loop terminates we annotate it with the following ranking function:

$$\rho = x - z.$$

Since the variable z is incremented during the loop the verification condition that establishes its termination will be as follows:

$$x - (z + 1) < x - z.$$

Showing Absence of Overflows

Five subgoals corresponding to the (sub-) expressions of the source code have to be proven in order to show the absence of run-time errors. These subgoals are:

$$\texttt{MIN\_INT} \leq x \leq \texttt{MAX\_INT},$$
$$\text{where } x \in \{2 \cdot z,\ y + 2 \cdot z,\ y + 2 \cdot z + 1,\ z + 1,\ x - 1\}.$$

All of them can be proven from the equation $x^2 \leq \texttt{MAX\_INT}$ of the precondition and from the invariant.

Chapter 4

Pointer Programs Verification

The string library involves pointer structures. Therefore we formalize in this chapter the idea of references and structures on them. Exploiting this formalization we present abstraction mapping relations between implementation and specification of pointer primitives.

4.1 References and Heap

The methodology of formal reasoning about complicated pointer data structures has been an open research topic for more then the last 30 years. The basic idea in all approaches goes back to Burstall [Bur72]. His model of memory made possible convincing proofs of list and tree algorithms. Refinements sufficient for the Hoare Logic are due to Bornat. In particular, in [Bor00] he proposes an idea of treating data structure components as a collection of variables. This allows to axiomatize the structure field assignments in the same way as variable assignments. Therefore the Hoare rules stay sufficient. The formalization of this representation is made by the team of Mehta and Nipkow [MN04]. They introduce *abstraction mappings* that interpret pointer structures as abstract data types. In this work we follow their ideas.

Nowadays every mainstream programming language supports *dynamic memory allocation*. This is the allocation of memory storage for use in a computer program during the run-time of that program. Memory is typically allocated from a large pool of all available unused memory called the *heap*. A dynamically allocated object remains in memory until it is deallocated explicitly, either by the programmer or by a garbage collector. We say that such an object has dynamic lifetime.

4.1.1 Pointer Structures

An access to a dynamic object is possible through a *pointer*, an element of the special data type whose value is used to *refer* (*point*) to another value. We use the term *reference* as a synonym for a pointer.

Dynamic memory provides a flexible mechanism for representing basic as well as complex data structures. Nodes in linked structures, e.g. linked lists and trees, are record-like objects. They consist of named components which we call *fields*. We distinguish two kinds of fields: fields containing *values* (value-fields) and fields containing *pointers* (pointer-fields).

Consider for instance the following C0 structure:

Listing 4.1: Structural type for a singly linked list

```
1 struct Node {
2   int          info;
3   struct Node  *next;
4 };
```

It represents a typical linked data structure — singly linked list (see section 4.2.2 for details). The field `info` is a value-field. Conversely, the field `next` is a pointer-field.

4.1.2 Formalization of Pointers

Our model of references is based on the two major principles. We support i) a simplified version of pointers without pointer arithmetic and ii) unbounded memory that can be allocated on the heap. To emphasize the former, pointers are not represented by numerical addresses, but we introduce a new abstract type $\mathbb{R}\text{ef}$. The latter prompts us to define this type isomorphic to natural numbers.

Definition 4.1 (Reference) *A* ***reference*** *is an element of the set* $\mathbb{R}\text{ef}$ *which is isomorphic to the set of natural numbers:*

$$\mathbb{R}\text{ef} \cong \mathbb{N}.$$

The element $Null \in \mathbb{R}\text{ef}$ *is isomorphic to* $0 \in \mathbb{N}$*:*

$$Null \cong 0.$$

As it follows from [Ste03] almost all approaches to formalize pointer structures exploit some kind of correspondence between the value of an object and its location in the heap. The location can be represented by a reference. We define the correspondence between references and object values by the *heap-function*. This idea follows Bornat [Bor00] and is suitable for encoding fields of records in C0.

Definition 4.2 (Heap-function) *A **heap-function** for a type $\mathbb{T}$ is a function drawn from the set*

$$H_{\mathbb{T}} \stackrel{\text{def}}{=} \{\delta \mid \delta : \mathbb{R}\text{ef} \to \mathbb{T}\},$$

such that for all $\delta \in H_{\mathbb{T}}$ and all pointers $p \in \mathbb{R}\text{ef}$, $\delta(p)$ yields the value which p points to.

We denote the set of all heap-functions, i.e. the functions from $\mathbb{R}\text{ef}$ to any type by

$$H_* \stackrel{\text{def}}{=} \bigcup_{\mathbb{T} \text{ is a type}} H_{\mathbb{T}}.$$

The two main properties of our model of heap-functions are:

1. Heap-functions are total, i.e. their application to any reference yields a certain value respecting types.

2. Heap-functions are uninterpreted in a sense that we do not know their return-values.

Thus, an application of heap-function implements an idea of *dereference*, an operation which yields the value an argument points to.

4.1.3 Encoding of the Record Fields

We encode the fields of records in C0 in the following way. Each node of a pointer structure is represented by a reference.

For each value-field of type $\mathbb{T}$ we introduce a heap-function drawn from the set $H_{\mathbb{T}}$. An application of this function to the reference yields the value of the respective field. This function is used to associate data with nodes of a pointer structure.

For each pointer-field we introduce a heap-function drawn from the set $H_{\mathbb{R}\text{ef}}$. Applying this function to the reference we obtain the some other node in the pointer structure. This function, therefore, is used to represent the links in the pointer structure.

Each of the generated functions we add to the state space Σ of a program which exploits a respective pointer structure. This can be viewed as an extension of the definition 3.1 which we agree on for the rest of this thesis.

Let us illustrate this with an example. Consider the structure from listing 4.1. We generate the following heap functions:

$$\begin{aligned} \mathit{info} &\in H_{\mathbb{Z}}, \\ \mathit{next} &\in H_{\mathbb{R}\text{ef}} \end{aligned}$$

for the fields `info` and `next` respectively.

Suppose that a reference to this structure is declared in the C0 program:

```
struct Node *node;
```

In our mathematical notation this corresponds to a variable from the state space:

$$node \in \mathbb{R}\text{ef}.$$

The access to the fields of a record pointed by `node` in the C0 code is as follows:

```
node->info
node->next
```

Exploiting our notation we can access the values of this fields applying the respective heap-functions. We, therefore, obtain:

$$\begin{aligned} info(node) &\in \mathbb{Z}, \\ next(node) &\in \mathbb{R}\text{ef}. \end{aligned}$$

The assignment to the fields of records in C0:

```
node->info := expr_1;
node->next := expr_2;
```

in the mathematical notation corresponds to the heap-function update (definition 2.8):

$$\begin{aligned} info &= info[node \to expr_1], \\ next &= next[node \to expr_2]. \end{aligned}$$

We assume that $expr_1$ and $expr_2$ are expressions which respect types.

4.1.4 Memory Allocation

The run-time creation of objects requires a function that yields a new reference to an object. Recall section 2.2.3, in the C0 programming language this function is implemented as the statement `new`. On the level of specification we define in the spirit of [Sch05] the function *new*.

Definition 4.3 (New) *Let A be a set of references. A function*

$$new : 2^{\mathbb{R}\text{ef}} \to \mathbb{R}\text{ef}$$

yields a non-zero reference, such that this reference is not contained in A:

$$new(A) \stackrel{\text{def}}{=} \varepsilon(\mathbb{R}\text{ef} \setminus (A \cup \{Null\})).$$

Recall that ε is the Hilbert's choice operator (definition 2.6).

For the bookkeeping of allocated references we support a special variable *alloc* of type $L^*_{\mathbb{R}\mathrm{ef}}$. We extend the state space Σ of every program with *alloc*. The abstract list *alloc* stores all pointers allocated during the run-time of a program. Clearly, a newly allocated reference must not be contained in the set $\mathcal{S}(alloc)$. To support this fact we agree on the following convention:

1. For the dynamic allocation of objects we always pass the set $\mathcal{S}(alloc)$ as an argument for the function *new*.

2. Each time we allocate a pointer $p = new(\mathcal{S}(alloc))$ we concatenate p with *alloc*: $alloc := p \odot alloc$.

This mechanism is depicted in figure 4.1.

4.2 Abstraction Mappings

The natural method to prove the correctness of complex data structures is as follows. At first one decomposes the structure into a variety of primitives, i.e. low-level structures. Then operations on these low-level structures are verified separately. Finally, the primitives are combined into the initial structure. Having correct primitives and a proved lemmata about their behavior makes the verification of the complex structure relatively easier. It is sufficient to prove only the correctness of the interaction of these verified primitives.

The general approach to verifying low-level structures is *abstraction*, i.e. mapping them to higher-level concepts. Following [MN04] we define several abstractions which play role of building blocks in more involved models, e.g. our model of strings. These building blocks we use for the specification of functions implementing algorithms for linked structures.

To introduce these abstractions we need an idea of the implementation and specification of a primitive. By the implementation of a primitive we understand some structure defined over the variables and heap-functions. By the specification of a primitive we understand an instance of a certain abstract data type, e.g. a tree or an abstract list whose elements are references. The concrete examples of these notions follow with sections 4.2.1, 4.2.2, and 4.2.3. For this chapter we will sometimes refer to implementation and specification of a primitive simply as to implementation and specification. The reader should not confuse them with implementation and specification of programs. Actually, we will use the former in specifications (pre- and postconditions) of programs.

Our key idea of abstraction mappings is to create a relation between the implementation and the specification of a primitive. We model the implementation of linked structures that we cover in this thesis as a sequence of references obtained by a consecutive application of heap-functions. The

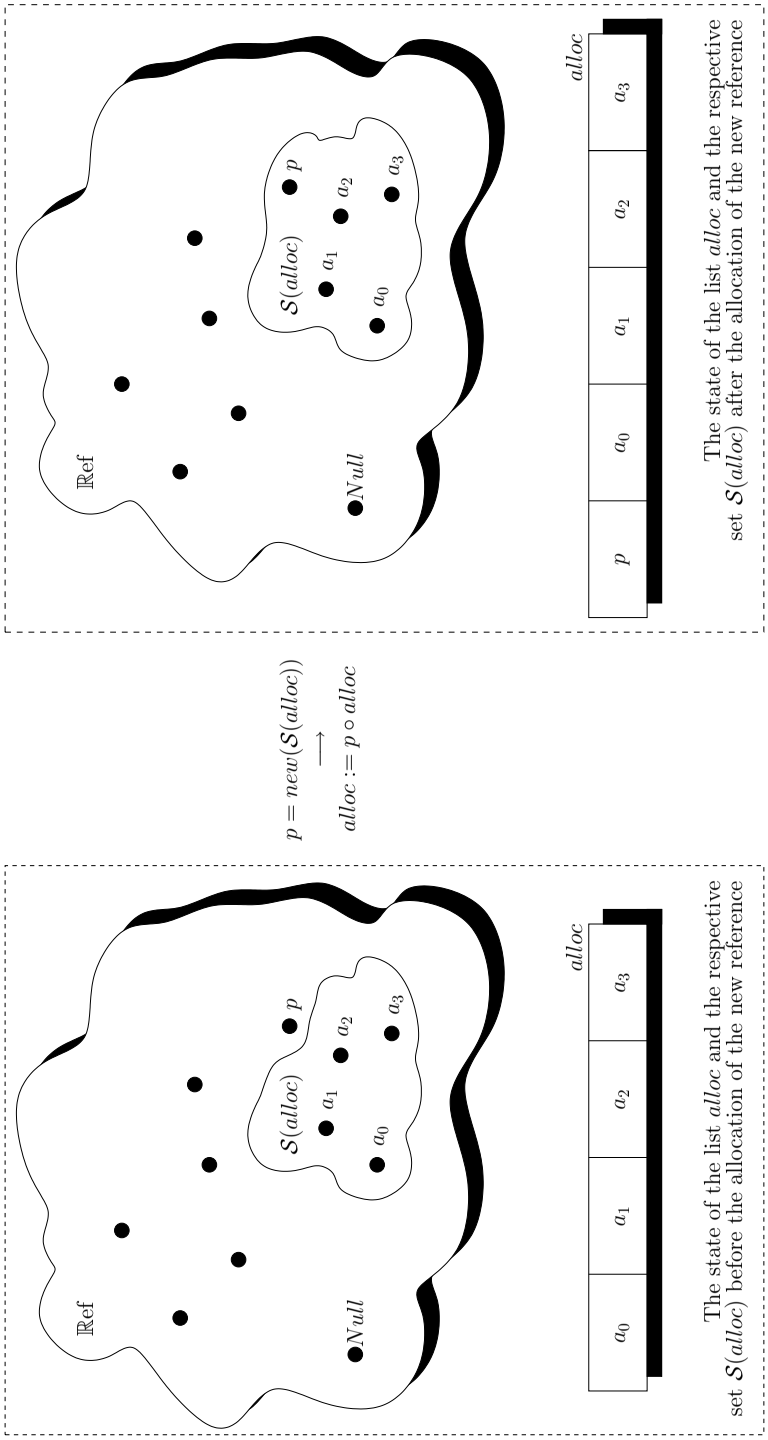

Figure 4.1: The mechanism for bookkeeping of allocated references

specification counterpart we represent as an abstract list of references. Then we relate both sequences by means of a predicate which returns True if the implementation meets the specification, and False otherwise. This idea is concreted in the sections below.

4.2.1 Path in the Heap

We start with the model of elementary segments of pointer structures. We will call them paths in the heap. The implementation of a pointer structure can be defined with the help of a heap-function $\delta \in H_{\mathbb{R}\mathrm{ef}}$ as follows. Consider a directed graph:

$$G = (V = \mathbb{R}\mathrm{ef}, E \subseteq V \times V).$$

This graph directly corresponds to the binary relation associated with the function δ. Its vertexes are references and edges are distinct pairs of references. Edges of G represent the "points to" relation. V contains the special vertex $Null$. The out-degree of this node is zero. The out-degree of each other node is exactly one, i.e. a reference cannot point to several different locations.

A *path in the heap* we call a path (in graph-theoretical sense) between two nodes of G. Clearly, the nodes of such a path can be specified by an abstract list of references $l \in L^*_{\mathbb{R}\mathrm{ef}}$.

For the implementation side consider a heap-function $\delta \in H_{\mathbb{R}\mathrm{ef}}$. Clearly, nodes of G are closed under application of this function. Therefore a successive application of δ started in the initial node $x \in V$ generates a sequence of nodes:

$$x,\ \delta(x),\ \delta^2(x),\ \ldots,\ \delta^n(x).$$

We are interested in the following property. Given two nodes $x, y \in V$ and a heap function $\delta \in H_{\mathbb{R}\mathrm{ef}}$ does the successive application of δ started in x end in y and yield a specified sequence $l[0], l[1], \ldots, l[n-1]$ of nodes of a pointer structure. Note that n is the number of function applications. Intuitively, this property relates the implementation of a pointer structure with its specification.

To use this property in the program specifications we express it as an assertion. Thus, formally we define a path in the heap as follows:

Definition 4.4 (Path in the Heap) *Let x and y be references, let δ be a heap-function for the type $\mathbb{R}\mathrm{ef}$, and let l be a list of references. The predicate*

$$Path : \mathbb{R}\mathrm{ef} \times H_{\mathbb{R}\mathrm{ef}} \times \mathbb{R}\mathrm{ef} \times L^*_{\mathbb{R}\mathrm{ef}} \to \mathbb{B}$$
$$Path(x, \delta, y, l) \stackrel{\mathrm{def}}{=} (\forall 0 \le i < |l|.\ \delta^i(x) = l[i] \wedge l[i] \neq Null) \wedge \delta^{|l|}(x) = y$$

yields True *iff l is a sequence of references that connects x to y by means of the heap-function δ.*

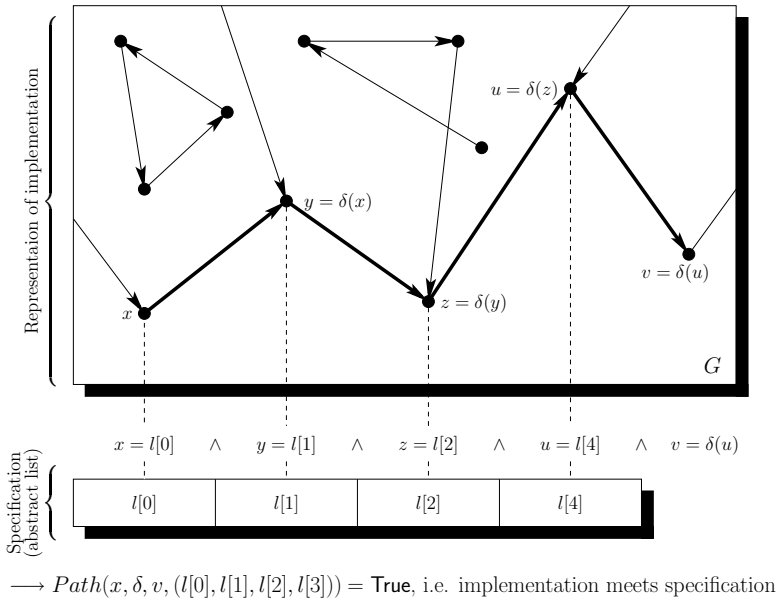

Figure 4.2: Relational abstraction of the path in the heap

Recall that $\delta^i(x)$ is a self composition of the δ function (definition 2.7).

Let us illustrate this notion with an example. Consider figure 4.2. Assume the heap-function $\delta \in H_{\mathbb{R}\text{ef}}$ with the following properties:

$$\begin{aligned} y &= \delta(x), \\ z &= \delta(y), \\ u &= \delta(z), \\ v &= \delta(u). \end{aligned}$$

Then, for example:

$$\begin{aligned} Path(x, \delta, v, (l[0], l[1], l[2], l[3])) &= \mathsf{True}, \\ Path(x, \delta, y, l[0]) &= \mathsf{True}, \\ Path(v, \delta, z, (l[2], l[1])) &= \mathsf{False}. \end{aligned}$$

4.2.2 Singly Linked List

Linked list is one of the fundamental data structures used in computer programming. It consists of a sequence of nodes. Each node contains arbitrary data fields and one or two references which point to the next (and/or previous) node. A linked list is called a *self-referential* data type because it contains a pointer to another instance of the same type. Several different types of linked list exist: singly linked lists, doubly linked lists, and circularly linked lists.

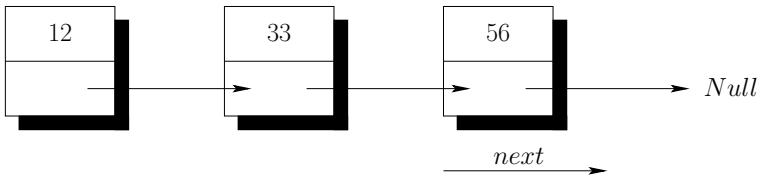

Figure 4.3: Singly linked list containing three integers

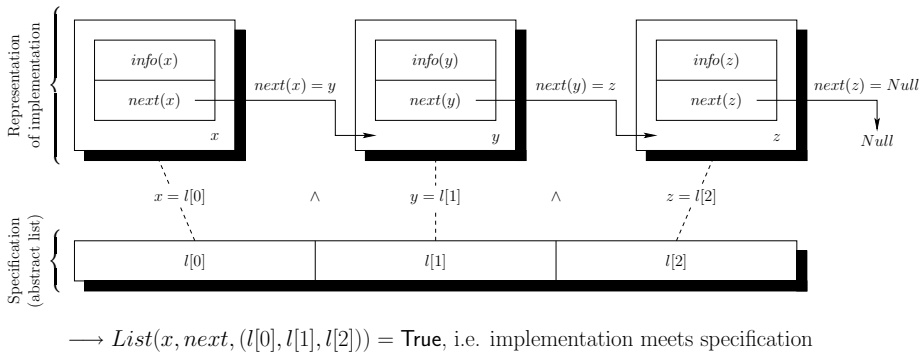

Figure 4.4: Singly linked list relational abstraction

A singly linked list has one pointer to an instance of the same type per node. This reference points to the next node in the list, or to a $Null$ value if it is the last node. The concept of a singly linked list is depicted in figure 4.3, a typical C0 structure for it is presented in listing 4.1.

Assume the nodes of a singly list are connected by means of heap-function $next : H_{\mathbb{R}\text{ef}}$. Therefore, such a list is nothing but a path in the heap ending in $Null$ with respect to the function $next$:

Definition 4.5 (Singly Linked List) *Let x be a reference, let $next$ be a heap-function for the type $\mathbb{R}$ef, and let l be a list of references. The predicate*

$$List : \mathbb{R}\text{ef} \times H_{\mathbb{R}\text{ef}} \times L^*_{\mathbb{R}\text{ef}} \to \mathbb{B}$$
$$List(x, next, l) \stackrel{\text{def}}{=} Path(x, next, Null, l)$$

yields True *iff l is the specification of a path in the heap from x to $Null$ with respect to the function $next$.*

As an example consider figure 4.4. Suppose $next \in H_{\mathbb{R}\text{ef}}$ is the heap-function with the following properties:

$$\begin{aligned} y &= next(x), \\ z &= next(y), \\ next(z) &= Null. \end{aligned}$$

In case the references on the implementation side are equal to the specified references, i.e.

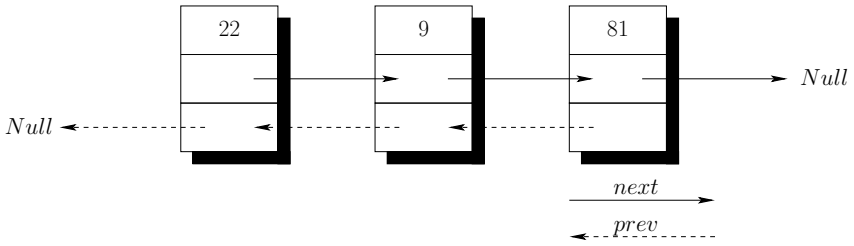

Figure 4.5: Doubly linked list containing three integers

$$x = l[0] \ \wedge \ y = l[1] \ \wedge \ z = l[2],$$

the predicate $List(x, next, (l[0], l[1], l[2]))$ yields True.

Equivalent recursive definitions of the *Path* and *List* abstractions can be found in [Sch04]. A variety of their properties is established by Schirmer and the accumulated lemmata are presented in [Sch04] as well.

4.2.3 Doubly Linked List

A more sophisticated kind of linked list is a doubly linked list (see figure 4.5). Each node has two references, one to the previous node and one to the next. Doubly linked lists require more space per node, and their elementary operations consume more machine time. On the other hand, they are often easier to manipulate because they allow sequential access to the list in both directions. The following C0 structure implements the doubly linked list.

Listing 4.2: Structural type for a doubly linked list

```
1 struct Node {
2   int           info;
3   struct Node   *next;
4   struct Node   *prev;
5 };
```

Since each node of a doubly linked list has two references its abstraction exploits two heap-functions. One of them applied to some node of a list returns its successor, the other — its predecessor. In this work we will denote this functions by *next* and *prev*, respectively.

Analogously to a path in the heap and a singly linked list, a doubly linked list can be specified by an abstract list l of pointers. The references on the implementation side can be obtained by the repeated application of heap-functions *next* and *prev* — the first should yield list l and with the second we should obtain the reversed list l^{-1}. Formally:

Definition 4.6 (Doubly Linked List) *Let x and y be references, let next and prev be heap-functions for the type* $\mathbb{R}$ref*, and let l be a list of references. The predicate*

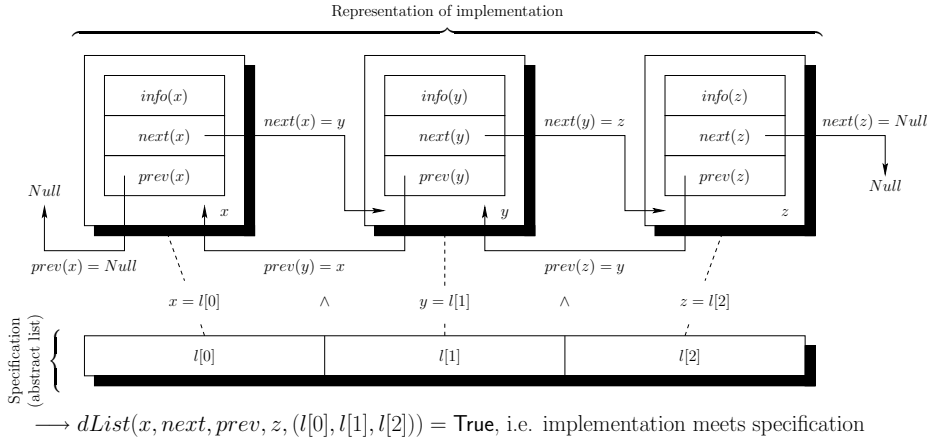

Figure 4.6: Doubly linked list relational abstraction

$$dList : \mathbb{R}\text{ef} \times H_{\mathbb{R}\text{ef}} \times H_{\mathbb{R}\text{ef}} \times \mathbb{R}\text{ef} \times L^*_{\mathbb{R}\text{ef}} \to \mathbb{B}$$
$$dList(x, next, prev, y, l) \stackrel{\text{def}}{=} List(x, next, l) \wedge List(y, prev, l^{-1})$$

yields True *iff* l *is the specification of a singly linked list started in* x *with respect to the function* $next$ *and* l^{-1} *is the specification of a singly linked list started in* y *with respect to the function* $prev$*.*

An example follows with figure 4.6. Let $next \in H_{\mathbb{R}\text{ef}}$ and $prev \in H_{\mathbb{R}\text{ef}}$ be heap-functions with the following properties:

$$\begin{aligned} x &= prev(y), \\ y &= next(x) = prev(z), \\ z &= next(y), \\ next(z) &= prev(x) = Null. \end{aligned}$$

If the references on the implementation side are equal to the specified references, i.e.

$$x = l[0] \ \wedge \ y = l[1] \ \wedge \ z = l[2],$$

the predicate $dList(x, next, prev, z, (l[0], l[1], l[2]))$ yields True.

The C0 implementation of doubly linked lists, the implementation of basic algorithms on them, and their formal verification are presented in [Ngu05].

Chapter 5

Implementation and Formal Specification of the String Library

We present implementation and specification of the target library in this chapter. For the implementation side we give data structure and functions which realize the desired algorithms on strings. We begin the specification side with an abstraction mapping for strings. We give formal specifications of functions and invariants as well ass ranking functions for their loops.

5.1 Overview of the Library

As it was mentioned in the introduction to this thesis, the work we present here is a part of the data structures and algorithms package which is being developed as a part of a formally verified complete computer system in the frame of the Verisoft project. The formally verified data structures and algorithms package will consist of the following libraries:

1. **Lists**
   This library provides support for doubly linked lists over arbitrary data types. By now it is finished, i.e. implemented and formally proved with respect to the partial correctness criteria. The results of this work are presented in [Ngu05].

2. **Strings**
   This library provides algorithms and data structures for strings, i.e. sequences of characters of an arbitrary length. A part of the algorithms was implemented and the partial correctness of these implementations was established in [Pre05]. The rest is due to the work presented in this thesis.

3. **Big Numbers**
   This library supports integers of arbitrarily large values. It is implemented and currently being verified.

4. **Directed Graphs**
   This library provides data structures and algorithms for directed graphs. It is implemented and currently being verified.

The string library consists of the following functions which implement the desired algorithms on strings:

1. **stringT**
   Creates a new empty string.

2. **stringAppend**
   Appends two strings.

3. **stringAppendChar**
   Appends a character to a string.

4. **stringGetChar**
   Gets a character at a certain position of a string.

5. **stringLength**
   Computes the length of a string.

6. **stringDeleteChar**
   Deletes a character at a certain position of a string.

7. **stringInsertChar**
   Inserts a character at a certain position of a string.

8. **subString**
   Extracts a substring from a string.

9. **stringCopy**
   Copies a string.

10. **stringEqual**
    Compares two strings lexicographically.

11. **stringFind**
    Searches a string for a given substring.

The target functions of this thesis are the last six functions.

5.2 Building Blocks

All the function in the library use some common building blocks, e.g. data types, constants. We start with their description.

5.2.1 Constants

For the whole library we define the following constants:

Listing 5.1: C0 definitions of constants for the library

```
1  #define NO_ERROR                0
2  #define ERROR_OUT_OF_MEMORY    -1
3  #define ERROR_OUT_OF_BOUNDS    -2
4  #define ERROR_INVALID          -3
5  #define ERROR_NOT_FOUND        -4
6  #define ERROR_DIV_BY_ZERO      -5
7  #define EQUAL                   0
8  #define LESS                    1
9  #define GREATER                 2
10
11 #define BLOCK_SIZE             16u
12 #define HALF_BLOCK_SIZE         8u
```

The constants defined in lines 1-6 are basically used as return codes of functions. They signal different kinds of errors or their absence. The constants from lines 7-9 are used to handle the result of the lexicographical comparison of strings. The constant `BLOCK_SIZE` defines the block size in the string data structure (see the next section for details). The constant `HALF_BLOCK_SIZE` is introduced due to the `stringInsertChar` function — we need to copy `BLOCK_SIZE`/2 characters in a branch of the algorithm, but we do not want to use the relatively complex division operation in the formalization of the function and its correctness proof.

5.2.2 Data Structures

The main design requirement for the string data structure is the ability of dynamic changes of string length. Since C0 programming language supports only fixed-size arrays this simple data structure does not suffice. Therefore we implemented a *growable array* as a data structure for strings in the following way. The elementary building block for this data structure is a fixed-size array with elements of type `char`. We define the size of such an array as `BLOCK_SIZE`. We create a pair consisting of such array and a variable `len` which stores the number of valid characters in the array. Then we organize a doubly linked list of such pairs. We refer to the elements of this doubly linked list as to *blocks*. The usage of linked lists allows us to change the string, e.g. add or delete characters dynamically. When an "add character" operation detects that there is no free space in the last block of a string it simply creates a new additional block and works with it. Conversely, when a delete operation removes the only remaining character from some block inside the string it deletes the obtained empty block. Additionally we support a special structure which contains a pointer to the first block. We

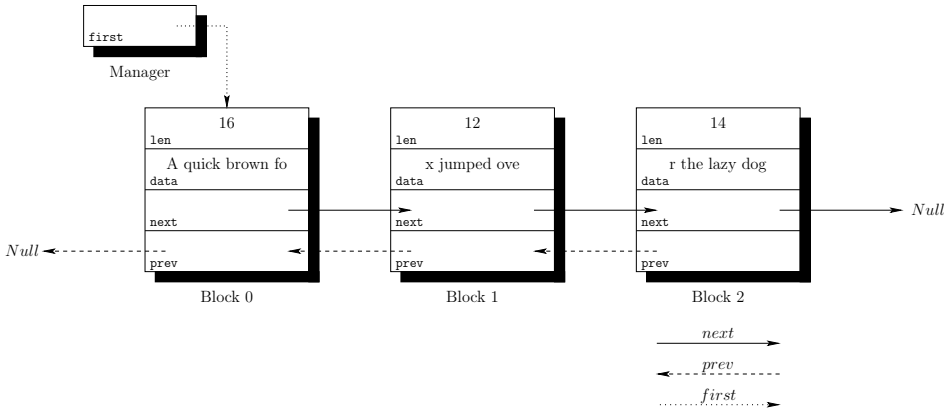

Figure 5.1: String containing three blocks

call this structure *manager*. The concept of our string implementation is depicted in figure 5.1.

The C0 implementation of this idea is presented in listing 5.2. The structure `string_block` directly corresponds to the elements of a doubly linked lists which we call blocks. The structure `string_mng` implements string manager. Additionally we provide an alias `string` for a pointer to the string manager.

Listing 5.2: C0 declaration of data structures for the library

```
1  /* Block of a string  */
2  struct string_block {
3    struct string_block   *next;
4    struct string_block   *prev;
5    unsigned int           len;
6    char                   data[BLOCK_SIZE];
7  };
8
9  /* Manager of blocks  */
10 struct string_mng {
11   struct string_block   *first;
12 };
13
14 /* Alias for the string type */
15 typedef struct string_mng *string;
```

5.2.3 String Abstraction

We introduce the following heap-functions on the side of specification to encode the fields of data structures for strings. The heap-functions

$$\begin{aligned} next &\in H_{\mathbb{R}\mathrm{ef}}, \\ prev &\in H_{\mathbb{R}\mathrm{ef}}, \\ len &\in H_{\mathbb{N}}, \\ data &\in H_{L_{\mathbb{C}\mathrm{har}}^{\texttt{BLOCK\_SIZE}}} \end{aligned}$$

correspond to the fields `next`, `prev`, `len`, and `data` of the structure `string_block`, respectively. The first three of them are straightforward. As for the last we represent the fixed size arrays of type `char` on the specification side as abstract lists of characters of fixed length (`BLOCK_SIZE`). Recall that we represent characters as elements of the set $\mathbb{C}\mathrm{har}$. Therefore in our notation such a list is an element of the set $L_{\mathbb{C}\mathrm{har}}^{\texttt{BLOCK\_SIZE}}$. The heap-function for this type is $H_{L_{\mathbb{C}\mathrm{har}}^{\texttt{BLOCK\_SIZE}}}$.

The heap-function

$$first \in H_{\mathbb{R}\mathrm{ef}}$$

corresponds to the field `first` of the structure `string_mng`.

For the rest of the thesis let Σ_0 denotes the common part of the state spaces of all functions from the library containing their heap-functions and the list *alloc*:

$$\begin{aligned} \Sigma_0 = (next &: H_{\mathbb{R}\mathrm{ef}}, \\ prev &: H_{\mathbb{R}\mathrm{ef}}, \\ len &: H_{\mathbb{N}}, \\ data &: H_{L_{\mathbb{C}\mathrm{har}}^{\texttt{BLOCK\_SIZE}}}, \\ first &: H_{\mathbb{R}\mathrm{ef}}, \\ alloc &: L_{\mathbb{R}\mathrm{ef}}^{*}). \end{aligned}$$

In analogy to the primitives defined in section 4.2 we introduce relational abstraction for strings. We specify strings by two abstract lists — one represents the structure of the string, the other represents the content. Since the formal definition of a string relation is a little bit involved we give its interpretation beforehand.

The *structure* of the string can be specified by the list l of references which corresponds to the specification of a doubly linked list (recall definition 4.6). The *content* of the string can be specified by the list s of characters. Sometimes we will refer to this list as *abstract string*. The relation between the implementation and specification must reflect both structure and content. Therefore the implementation of the string is correct with respect to its specification if the following conditions are fulfilled. Assume that there is a pointer p to the string manager, then:

1. There exists a doubly linked list started in $first(p)$ with respect to the heap-functions $next$ and $prev$ which is specified by the list l, formally:

$$\exists q.\, dList(first(p), next, prev, q, l).$$

2. The concatenation of all valid characters in all blocks yields the list s. Formally:

$$s = \bigodot_{0 \leq i < |l|} data(l[i])[0 : len(l[i]) - 1].$$

The additional requirements in the string abstraction are mostly technical. We require:

3. The pointer p to the string manager is not $Null$.

4. The number of valid characters in any block is a positive natural number upper bounded by `BLOCK_SIZE`. Formally:

$$\forall 0 \leq i < |l|.\, 0 < len(l[i]) \leq \texttt{BLOCK\_SIZE}.$$

The formal definition of the string relational abstraction is the conjunction of the four requirements above.

Definition 5.1 (String) *Let p be a reference, let $first$, $next$, and $prev$ be heap-functions for the type $\mathbb{R}\mathrm{ef}$, let len be a heap-function for the type $\mathbb{N}$, let $data$ be a heap-function for the type $L^{\texttt{BLOCK\_SIZE}}_{\mathbb{C}\mathrm{har}}$, let l be a list of references, and let s be a list of characters. The predicate*

$$String : \mathbb{R}\mathrm{ef} \times H_{\mathbb{R}\mathrm{ef}} \times H_{\mathbb{R}\mathrm{ef}} \times H_{\mathbb{R}\mathrm{ef}} \times H_{\mathbb{N}} \times H_{L^{\texttt{BLOCK\_SIZE}}_{\mathbb{C}\mathrm{har}}} \times L^{*}_{\mathbb{R}\mathrm{ef}} \times L^{*}_{\mathbb{C}\mathrm{har}} \to \mathbb{B}$$

is such that

$$\begin{array}{c} String(p, first, next, prev, len, data, l, s) \\ \stackrel{\text{def}}{=} \\ (\exists\, q\, .\, dList(first(p), next, prev, q, l)) \\ \wedge\, s = \bigodot_{0 \leq i < |l|} data(l[i])[0 : len(l[i]) - 1] \\ \wedge\, p \neq Null \\ \wedge\, \forall\, 0 \leq i < |l|\, .\, 0 < len(l[i]) \leq \texttt{BLOCK\_SIZE}. \end{array}$$

The essence behind this definition is depicted in figure 5.2. Suppose the properties of heap-functions are as follows:

$$\begin{array}{rcl} u & = & first(p) = prev(v), \\ v & = & next(u) = prev(q), \\ q & = & next(v), \\ next(q) & = & prev(u) = Null. \end{array}$$

Figure 5.2: String relational abstraction

$\longrightarrow String(p, first, next, prev, len, data, (l[0], l[1], l[2]), s) = \mathsf{True}$, i.e. implementation meets specification

If the references obtained by the application of heap-functions are equal to the specified structure of the string, i.e.

$$u = l[0] \wedge v = l[1] \wedge q = l[2],$$

and the concatenation of valid characters in all block yields the specification of the string content, i.e.

$$data(u)[0 : len(u) - 1] \odot data(v)[0 : len(v) - 1] \odot data(q)[0 : len(q) - 1] = s,$$

the predicate $String(p, first, next, prev, len, data, (l[0], l[1], l[2]))$ yields True.

5.3 General Notes on Implementation and Specification of the Functions

The rest of this chapter is organized as follows. For each of the six target functions we present their:

1. implementation,
2. formal specification,
3. loop invariants, and
4. loop ranking functions.

In this section we give some general remarks on each of the points.

5.3.1 Implementation

Concerning the implementation, its general requirement is to be low-level. This means that whenever it is possible, we implement the function in a way it works directly with the string's block structure, instead of calling already implemented operations on strings. This principle allows us to reduce the running time of the functions to *linear* in number of processed blocks [1]. For each of the functions we give its signature with the annotation of the formal parameters and return-results, sketch the algorithm we use, and give the implementation in the C0 programming language. The implementation details are given as well.

[1] The only exception is the function `stringFind` which runs in quadratic time due to the algorithm involved.

5.3.2 Specification

The formal specifications of the functions are given by their pre- and post-conditions. Additionally, for each of the functions we give an assertion called the *extensions for guards*. We need to include it in the precondition in order to prove the absence of run-time errors in a program.

For the presentation of the assertions for pre- and postconditions as well as for the extensions for guards we use the following naming convention:

$$x\_name,$$

where $x \in \{PRE, POST, GUARD\_EXT\}$ and *name* is the name of the function. For example,

$$PRE\_subString, POST\_subString, GUARD\_EXT\_subString \in \mathcal{A}_{\Sigma},$$

where Σ is the state-space of the function `subString`, are the pre- and postconditions, and the extensions for guards of the function `subString` respectively. The tuple

$$(PRE\_subString, POST\_subString)$$

forms the specification (not considering the run-time faults), when the tuple

$$(PRE\_subString \wedge GUARD\_EXT\_subString, POST\_subString)$$

gives the specification sufficient for the correctness proof as well as for the absence of run-time errors proof.

Precondition

The preconditions of the functions usually state the sufficient string relation. The string's structure and content are specified with abstract lists. Generally these lists will be called l_{PRE} and $s[0:n-1]$ respectively. We assume these lists to be in the global context of reasoning, i.e. they are visible not only in the precondition, but in the postcondition and the loop invariants as well. The next simplification touches the size of an abstract string. We give it exploiting the definition of an abstract list. In the example above this size is n. We suppose this parameter to be visible everywhere as well. Sometimes in order to increase expressivity of an assertion we give the abstract list size using the length notation — $|s|$.

Postcondition

Clearly, the variables from the state space Σ of a function can change their values from step to step of the program execution. Sometimes when we formulate postconditions we need to refer to the values of the variables that they had in preconditions. For this purpose we will supplement the name of a

variable with superscribed prefix 0. For example ${}^0alloc$ in the postcondition or invariant denotes the value of the abstract list *alloc* which it had in the state satisfying precondition. The non-prefixed version of a variable always corresponds to its value in the current state.

We will use this notation in the invariants as well.

Naturally, all the variables in a precondition equal their ^{0}-prefixed versions. We will not write this fact explicitly, but will assume it for the proof in chapter 6.

Separation Properties

Some of the functions from the library, namely `stringDeleteChar`, `stringInsertChar`, `subString`, and `stringCopy` modify the string during their execution. On the specification level these modifications boil down to the function update of the heap-functions involved into the specified string relation. Here the question arises of how one can ensure that string modification do not harm other data structures build on the same heap-functions. The properties of this nature are called the *separation properties*. Clearly, the function user would not like to prove such property every time the function is called. Thus, we have included into the specifications of the mentioned above functions the sufficient separation properties.

In order to formalize these properties we introduce the heap-function preservation predicate $Pres$. It receives a list of references and a set of heap-functions as arguments. It has the meaning that whenever a reference *does not belong* to the given list, the heap-functions from the given set *were not updated* at this reference. We put it formally (exploiting the set of all heap-function notation H_* — definition 4.2):

Definition 5.2 (Heap-Function Preservation) *Let l be the list of references, and let H be the set of heap-functions. The* ***heap-function preservation*** *predicate*

$$Pres : L^*_{\mathbb{R}\mathrm{ef}} \times 2^{H_*} \to \mathbb{B}$$
$$Pres(l, H) \stackrel{\mathrm{def}}{=} \forall x \notin \mathcal{S}(l)\,.\,\forall \delta \in H\,.\,{}^0\delta(x) = \delta(x)$$

yields True *iff all the heap functions from the set H have the same value that they had in the precondition of a function being applied to any reference which is not contained in the list l.*

For example the condition $Pres(l, \{next, prev\})$ is nothing but the abbreviation of the following formula:

$$\forall x \notin \mathcal{S}(l)\,.{}^0next(x) = next(x) \ \wedge \ {}^0prev(x) = prev(x).$$

According to our notational convention the ^{0}-prefixed variables denote a heap-function in the state satisfying precondition, the non-prefixed — in the state when the predicate $Pres$ is evaluated.

In order to state that none of the heap-functions from a set H is changed we will write $Pres([\,], H)$ because there is no reference that belongs to the empty list. Finally, to state that the heap-functions are preserved for all but one reference, we will pass this reference as the first parameter of the predicate $Pres$, implicitly assuming that it forms the list of one element.

5.3.3 Loop Invariants

The loop invariants are represented by assertions. We stick to the following naming convention for loop invariants:

$$INV\_x\_name,$$

where x is the loop number in the order they appear in the implementation, and *name* is the function's name. For example, $INV\_2\_stringCopy \in \mathcal{A}_\Sigma$ is the invariant for the second loop of the function `stringCopy`.

It is the case that sometimes the loops of the different functions have the common structure and meaning of the implementation. Therefore, the invariants for these loops will look similarly. In this case we describe such a loop in details once, and refer to it further when the analogous loop is considered.

5.3.4 Ranking Functions

Regarding the ranking functions, they are represented as objects of type $\mathcal{R}_\Sigma$, i.e. the functions from the state space to the naturals (recall definition 3.11). All the variables in their definitions are evaluated, therefore, in a state during the loop execution. The naming convention for the ranking functions is as follows:

$$RANK\_x\_name,$$

where x and *name* have the same meaning as above.

5.4 Function `stringDeleteChar`

This function is supposed to delete a single character from the string at a certain position. The specified signature of the functions is:

```
int stringDeleteChar(string p, unsigned int pos)
```

where `p` is the pointer to the string manager and `pos` is the position in the string at which a character must be deleted. The return values can be:

- `ERROR_OUT_OF_BOUNDS` signals that the position `pos` is greater than the size of the string. In this case the string pointed to by `p` must not be changed at all.
- `NO_ERROR` is returned if the function execution is finished successfully.

5.4.1 Implementation

The idea of the algorithm is as follows:

1. Find the block from which a character must be deleted.
2. Delete this character by shifting the remaining characters in this block one position left and decrementing the length of this block by one.
3. Delete this block from the string if it becomes empty.

The following C0 code implements the described algorithm:

Listing 5.3: Deletion of a character from the string

```
1  int stringDeleteChar(string p, unsigned int pos)
2  {
3    struct string_block *cb;   /* current block */
4    int                 res;  /* return-result */
5    unsigned int        size; /* length of the string */
6
7    res  = NO_ERROR;
8    size = stringLength(p);
9
10   /* Check whether pos is in interval [0..size-1] */
11   if (size <= pos) {
12     res = ERROR_OUT_OF_BOUNDS;
13   } else {
14     cb = p->first;
15
16     /* Find the block which corresponds to pos */
17     while (cb->len <= pos) {
18       pos = pos - cb->len;
19       cb  = cb->next;
20     }
21
22     /* Shift the data in the current block */
23     while (pos + 1u < cb->len) {
24       cb->data[pos] = cb->data[pos + 1u];
25       pos           = pos + 1u;
26     }
27
28     /* Decrement the length of the block by one */
29     cb->len = cb->len - 1u;
30
31     /* Delete the block in case it is empty */
32     if (cb->len == 0u) {
33       p->first = dlist_block_Delete(p->first, cb);
34     }
35   }
36
37   return res;
38 }
```

In lines 3-5 we declare variables for storing the current block (`cb`) of the string, the return result of the function (`res`), and the length of the string (`size`). In lines 7-8 and 14 we initialize them with respective values. We perform the check whether the position `pos` is greater than the length of the string and signal, if it is necessary, the respective error in lines 11-13. In lines 17-20 we continue with the search for the block from which a character has to be deleted. After the appropriate block is found the pointer to it is stored in `cb`. In lines 23-26 the characters in the current block are shifted one position left starting from `pos` up to the end of this block. Then the length of `cb` is decremented by one. We finish with the deletion of the current block from the string in case the length of this block is zero (lines 32-34). The deletion is performed with help of the function `dList_block_Delete` from the library for doubly linked lists.

5.4.2 Formal Specification

The state space over which we formulate the specification is:

$$\Sigma = \Sigma_0 \cup (p : \mathbb{R}\text{ef}, pos : \mathbb{N}, cb : \mathbb{R}\text{ef}, size : \mathbb{N}, res : \mathbb{Z}).$$

Precondition

As it follows from the informal specification we demand in the precondition the existence of the string from which a character must be deleted. Let an abstract list l_{PRE} be the specification of the string structure. The specification of the string content is s. Formally the precondition is:

PRE_stringDeleteChar

1 $String(p, first, next, prev, len, data, l_{PRE}, s[0{:}n{-}1])$

Postcondition

The postcondition branches on the result of the test whether the position pos is less then the length n of the specified string s. The result codes of function termination in these two cases are specified in lines 4 and 9 of the formal representation of the postcondition below. In both cases of function termination the string relation holds. In case of termination with `NO_ERROR` code the string is changed. Therefore we claim in line 2 the existence of an abstract list l. It is the specification of the string structure in this case. The desired changes on the string content results in the specification of content: $s[0 : pos - 1] \odot s[pos + 1 : n - 1]$. In case of termination with `ERROR_OUT_OF_BOUNDS` code we claim in line 7 that the initial string stays unchanged. Additionally we require in line 3 the elements of the resulting list l to be included in the set of elements of initial list l_{PRE}. Formally the postcondition is:

POST_stringDeleteChar

1 $(pos < n \longrightarrow$
2 $(\exists l.\, String(p, first, next, prev, len, data, l, s[0{:}pos{-}1] \odot s[pos{+}1{:}n{-}1])$
3 $\wedge\, \mathcal{S}(l) \subseteq \mathcal{S}(l_{PRE}))$
4 $\wedge\, res = \texttt{NO\_ERROR}$
5 $\wedge\, Pres(p, \{first\}) \wedge Pres(l_{PRE}, \{next, prev, len, data\}))$
6 $\wedge\, (pos \geq n \longrightarrow$
7 $String(p, first, next, prev, len, data, l_{PRE}, s[0{:}n{-}1])$
8 $\wedge\, Pres([\,], \{first, next, prev, len, data\})$
9 $\wedge\, res = \texttt{ERROR\_OUT\_OF\_BOUNDS})$

Note, that lines 5 and 8 state the separation property for the function's two outcomes. In case of successful deletion the heaps may change as follows. Since it could be the case that the first block of the string contains the single character to be deleted, which results in the deletion of the whole first block, we state that the heap-function $first$ is preserved for all references but p (first conjunct of line 5). Concerning the other heap-functions, they are preserved everywhere except for the list initial strings structure l_{PRE}. This is cased by the possible deletion of the block at any position of the string (second conjunct of line 5). Conversely, in case of the return from the function with an error, the heap function stay unchanged (line 8).

Extensions for Guards

The length of the string's content specification is upper bounded by the `MAX_NAT` constant:

GUARD_EXT_stringDeleteChar

1 $|s| \leq \texttt{MAX\_NAT}$

5.4.3 Loop Invariants

First Loop

In the first loop of this function (lines 17-20 of listing 5.3) we scan the doubly linked list of blocks in order to find the block from which a character has to be deleted. The assertion for the invariant of this loop is as follows:

INV_1_stringDeleteChar

1 $(\exists hd, tl.\, String(p, first, next, prev, len, data, hd \odot cb \odot tl, s)$
2 $\wedge\, {}^{0}pos = pos + \sum_{0 \leq i < |hd|} len(hd[i])$
3 $\wedge\, \mathcal{S}(hd) \cup \{cb\} \cup \mathcal{S}(tl) \subseteq \mathcal{S}(l_{PRE}))$
4 $\wedge\, cb \neq Null$
5 $\wedge\, Pres([\,], \{first, next, prev, len, data\})$
6 $\wedge\, res = \texttt{NO\_ERROR}$

In the formal representation of the loop invariant the scan of the string results in the partition of the string's structure into three parts:

1. The head list *hd*. Its intended meaning is to specify the sequence of references to the blocks which have already been traversed.

2. The current block *cb*.

3. The tail list *tl*. It is needed to specify the sequence of references to the blocks which are going to be traversed during the further loop execution.

Consider line 1 of the assertion above. The string's structure is given according to the partition scheme.

During the loop execution we modify the variable *pos*. On each iteration its value is decremented by the number of valid characters in the current block. Clearly, the sum of the lengths over all blocks traversed by the moment is the difference between the initial value ${}^0pos$ and the current value *pos*. Formally this fact is stated in line 2.

In line 3 we state the relation between the elements of the lists *hd* and *tl*, the reference *cb* and the initial specification list l_{PRE} of string structure (see precondition). Line 5 gives the separation property — the heap-functions do not change during this loop. Line 6 reflects the fact that loop execution does not change the value of the function return code.

Second Loop

In the second loop (lines 23-26 of listing 5.3) we shift one position left all the characters inside the current block which lay to the right of the character in the position ${}^0pos$.

There is a difference between the intended meaning of positions ${}^0pos$ and *pos*. The former denotes the initial value of the position counter (as it has in the state satisfying precondition). Recall line 18 of listing 5.3 — we use variable `pos` as a counter. Therefore *pos* denotes the current value of this counter, i.e. during the loop execution.

To state the invariant for this loop let us agree on the following shorthands for string intervals (all the notations respect the invariant for the first loop):

$$\begin{aligned}
a &= \sum_{0 \le i < |hd|} len(hd[i]), \\
b &= {}^0pos - \sum_{0 \le i < |hd|} len(hd[i]), \\
c &= pos - {}^0pos + \sum_{0 \le i < |hd|} len(hd[i]), \\
d &= len(cb) - pos, \\
e &= \sum_{0 \le i < |tl|} len(tl[i]).
\end{aligned}$$

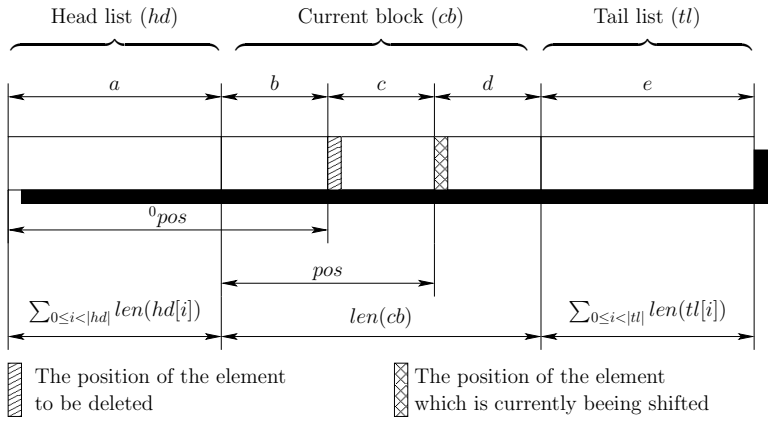

Figure 5.3: String slicing in the invariant for the second loop of the function `stringDeleteChar`

The slicing of the string with respect to these intervals is depicted in figure 5.3. Here ${}^0pos$ denotes the position in the string at which a character must be deleted. We distinguish it from pos which means the relative to the current block position of the character which is being shifted at the moment.

During the execution of the second loop the following assertion holds:

INV_2_stringDeleteChar

1 $(\exists hd, tl, a, b, c, d, e.\ String(p, first, next, prev, len, data, hd \odot cb \odot tl,$
2 $\qquad s[0\!:\!a\!+\!b\!-\!1] \odot s[a\!+\!b\!+\!1\!:\!a\!+\!b\!+\!c] \odot s[a\!+\!b\!+\!c\!:\!n\!-\!1])$
3 $\qquad \wedge\ s[0\!:\!a\!-\!1] = \bigodot_{0 \le i < |hd|} data(hd[i])[0\!:\!len(hd[i])\!-\!1]$
4 $\qquad \wedge\ s[a\!:\!a\!+\!b\!-\!1] \odot s[a\!+\!b\!+\!1\!:\!a\!+\!b\!+\!c] \odot s[a\!+\!b\!+\!c\!:\!a\!+\!b\!+\!c\!+\!d\!-\!1]$
5 $\qquad\qquad = data(cb)[0\!:\!len(cb)\!-\!1]$
6 $\qquad \wedge\ s[a\!+\!b\!+\!c\!+\!d\!:\!n\!-\!1] = \bigodot_{0 \le i < |tl|} data(tl[i])[0\!:\!len(tl[i])\!-\!1]$
7 $\qquad \wedge\ \mathcal{S}(hd) \cup \{cb\} \cup \mathcal{S}(tl) \subseteq \mathcal{S}(l_{PRE}))$
8 $\wedge\ Pres(cb, \{data\})\ \wedge\ Pres([\,], \{first, next, prev, len\})$
9 $\wedge\ res = \texttt{NO\_ERROR}$

In line 1 we state that there is a string relation. Its structure is specified by the abstract list $hd \odot cb \odot tl$, where hd, cb, and tl stand for head list, pointer to the current block, and tail list, respectively. The meaning of these entities is the same as in the invariant for the first loop. The content of the string is specified by the abstract list

$$s[0 : a + b - 1] \odot s[a + b + 1 : a + b + c] \odot s[a + b + c : n - 1].$$

It corresponds to the content s of the initial string where the two following changes are made:

1. The character in the position ${}^0pos$ is deleted. It is easy to see that $a + b = {}^0pos$. Therefore the concatenation $\ldots a + b - 1] \odot s[a + b + 1 \ldots$ specifies the deletion.

2. The character in the position $pos + \sum_{0 \leq i < |hd|} len(hd[i])$ is duplicated one position right. This reflects the progress of shifting characters inside the current block. One can check that $a + b + c = pos + \sum_{0 \leq i < |hd|} len(hd[i])$. Therefore the concatenation $\ldots a + b + c] \odot s[a + b + c \ldots$ specifies the character duplication.

In line 3 we state the equality between the implementation of the string part whose structure is specified by the list *hd* and the specification of its content. The implementation of this part is stated as the concatenation of all valid characters in all blocks pointed to by the elements of *hd*. The content specification of this part of the string is $s[0 : a - 1]$.

In line 4 we state the same kind of property for the current block. The implementation counterpart is $data(cb)[0 : len(cb) - 1]$, i.e. all the valid characters in the current block. The specification of this interval is a part of the string content specification (recall line 2) where the parts which correspond to the head and tail lists are dropped.

In line 6 we claim a similar property for the tail list. Line 8 shows the fact that *data* was changed only in the current block, and all the other heap-functions are complete preserved. The reasoning behind the remaining lines is the same as in the invariant for the first loop.

5.4.4 Ranking Functions

First Loop

In the line 18 of the function's implementation listing we decrement the variable *pos* — with every iteration of the loop it becomes smaller. Therefore, it suffices to be the ranking function:

RANK_1_stringDeleteChar

1 pos

Second Loop

During this loop the variable *pos* is incremented while it is less than the length of the current block (line 23 and 25 of listing 5.3). The ranking functions is therefore as follows:

RANK_2_stringDeleteChar

1 $len(cb) - pos$

5.5 Function `stringInsertChar`

The intended behavior of this function is to insert a single character into the string at a given position. The signature is:

```
int stringInsertChar(string p, char ch, unsigned int pos)
```

where `p` is the pointer to the string manager and `ch` is the character to be inserted at the position `pos` of the string. The termination of this function is possible with the following return codes:

- `ERROR_OUT_OF_BOUNDS` occurs in case `pos` is greater than the size of the string. The string therefore stays unchanged.
- `ERROR_OUT_OF_MEMORY` signals that the function has ran out of memory. The string in this case stays unchanged.
- `NO_ERROR` is the code of successful termination.

5.5.1 Implementation

We sketch the algorithm of the function:

1. Find the block into which a character must be inserted.
2. Determine whether the current block is full and:
   (a) in case it is, create the new empty block, copy the `HALF_BLOCK_SIZE` last characters from the current block into the new block, and insert the new block into the string;
   (b) otherwise increase the number of valid characters in the current block by one.
3. Insert the character into the current block by shifting the remaining characters in this block one position right and assigning the desired value to the character in the desired position.

Obviously, in the second step it is necessary to copy only the one character to the new block in case the current block is full. It turns out that this is not the best idea because of the following. imagine a situation when the user of the function inserts the character into the same position repeatedly. Clearly, at some point of time the block in which the character is inserted becomes full. Further, every insertion of the character will produce the new block in the string which contains only this character. In this scenario the string will contain unnecessarily many blocks (filled only with one character). This will decrease the running time of the algorithms on the string (caused by the redundant pointer dereferencing). In order to prevent this drawback we copy `HALF_BLOCK_SIZE` characters in the second step of the algorithm.

The source code of the C0 implementation of this algorithm is as follows:

Listing 5.4: Insertion of a character into the string

```
1  int stringInsertChar(string p, char ch, unsigned int pos)
2  {
3    unsigned int         i,    /* counter */
4                         size; /* string's length */
5    struct string_block *cb,   /* current block */
6                        *nb;   /* new block */
7    int                  res;  /* return-result */
8
9    size = stringLength(str);
10   res  = NO_ERROR;
11
12   if (size <= pos) {
13     res = ERROR_OUT_OF_BOUNDS;
14   } else {
15     cb = p->first;
16
17     /* Find the block which corresponds to pos */
18     while (cb->len <= pos) {
19       pos = pos - cb->len; cb = cb->next;
20     }
21
22     if (cb->len == BLOCK_SIZE) {
23       nb = new(struct string_block);
24       if (nb == NULL) {
25         res = ERROR_OUT_OF_MEMORY;
26       } else {
27         i = 0u;
28
29         /* Copy second half into the new block */
30         while (i + HALF_BLOCK_SIZE < BLOCK_SIZE) {
31           nb->data[i] = cb->data[i + HALF_BLOCK_SIZE]; i = i + 1u;
32         }
33
34         cb->len = HALF_BLOCK_SIZE; nb->len = HALF_BLOCK_SIZE;
35
36         /* Insert new block into the string */
37         res = dlist_block_InsertAfter(cb, nb);
38
39         /* Check whether pos is in the second half */
40         if (res == NO_ERROR && HALF_BLOCK_SIZE <= pos) {
41           cb = cb->next; pos = pos - HALF_BLOCK_SIZE;
42         }
43       }
44     }
45
46     if (res == NO_ERROR) {
47       cb->data[cb->len] = cb->data[cb->len - 1u];
48       i                 = cb->len - 1u;
49       cb->len           = cb->len + 1u;
50
51       /* Shift the data in the current block */
```

```
52        while (pos < i) {
53          cb->data[i] = cb->data[i - 1u]; i = i - 1u;
54        }
55        cb->data[pos] = ch;
56      }
57    }
58
59    return res;
60  }
```

The function implementation starts (in lines 3-7) with the declaration of variables for storing the pointers to the current block (`cb`) and to the new block (`nb`), the length of the string (`size`), the return result of the function (`res`), and the auxiliary counter (`i`). In lines 9-10 we assign to `size` and `res` their intended initial values. In lines 12-13 we check whether the position is greater than the length of the string and if it is necessary we signal `ERROR_OUT_OF_BOUNDS`.

Along lines 15-20 we search for an appropriate block in which a character must be inserted. When such block is found we test whether it is full (line 22). In case it is, we allocate a chunk of memory sufficient for an instance of the `string_block` structure and store a pointer to this memory in the variable `nb` (line 23). In the case of unsuccessful memory allocation the corresponding error flag is risen (lines 24-25). We continue in lines 29-32 by the copying of the second half of characters from the current block `cb` into the new block `nb`, setting the lengths of the blocks to `HALF_BLOCK_SIZE`, and insertion the new block into the string. The last is performed by the function `dList_block_InsertAfter` from the doubly linked lists package.

As the next step we determine into which block the character must be inserted: the current of the next one. This directly corresponds to the test whether `pos` lies in the first or the second half of the current block (line 40). In case it lies in the second half, it was copied to the new block. After its insertion into the string it becomes the next block relative to the current one. Therefore the sufficient step in the block structure is done in line 41.

In line 47 the last character in the current block is shifted one position right. We handle this character separately since it eases verification of the function. Further we shift the other characters in this block in order to obtain a free slot in the block for the character to be inserted (lines 51-54). Finally, the assignment of the desired value `ch` to the character at the position `pos` finishes the implementation.

5.5.2 Formal Specification

The state space over which we formulate the specification is:

$$\Sigma = \Sigma_0 \cup (p : \mathbb{R}\text{ef}, ch : \mathbb{C}\text{har}, pos : \mathbb{N}, cb : \mathbb{R}\text{ef}, nb : \mathbb{R}\text{ef}, \\ size : \mathbb{N}, i : \mathbb{N}, res : \mathbb{Z}).$$

Precondition

The reasoning behind the precondition is similar to the function `stringDeleteChar`. The additional requirement on elements of the list l_{PRE} and the reference p to be contained in the set $\mathcal{S}(alloc)$ is caused by the fact that this function is supposed to allocate new memory. Formally the precondition is:

PRE_stringInsertChar

1 $String(p, first, next, prev, len, data, l_{PRE}, s[0{:}n{-}1]) \wedge \mathcal{S}(l_{PRE}) \cup \{p\} \subseteq \mathcal{S}(alloc)$

Postcondition

The postcondition is formulated similarly to the postcondition of the function `stringDeleteChar`. In line 2 below we state the existence of the resulting string relation. The content of the string is specified by the abstract list of characters: $s[0 : pos-1] \odot ch \odot s[pos : n-1])$. The condition in line 3 states that the inserted element, if there is such, is allocated. The condition in line 4 requires the reference to the new block in case of its insertion not to be contained in the initial set $\mathcal{S}({}^0alloc)$. Line 5 gives the separation property, it is analogical to the one claimed in the postcondition of the previous function. Line 6 reflects the fact that the set $\mathcal{S}(alloc)$ grows in case of a character insertion. Line 10 states the constancy of $alloc$ in case of termination with `ERROR_OUT_OF_BOUNDS` code, while line 11 gives the separation property for this case.

Formally the postcondition is:

POST_stringInsertChar

1 $(pos < n \longrightarrow$
2 $(\exists l.\, String(p, first, next, prev, len, data, l, s[0{:}pos{-}1] \odot ch \odot s[pos{:}n{-}1])$
3 $\wedge\, \mathcal{S}(l) \setminus \mathcal{S}(l_{PRE}) \subseteq \mathcal{S}(alloc)$
4 $\wedge\, (\mathcal{S}(l) \setminus \mathcal{S}(l_{PRE})) \cap \mathcal{S}({}^0alloc) = \varnothing$
5 $\wedge\, Pres(p, \{first\}) \wedge Pres(l, \{next, prev, len, data\}))$
6 $\wedge\, \mathcal{S}({}^0alloc) \subseteq \mathcal{S}(alloc)$
7 $\wedge\, res = \texttt{NO\_ERROR})$
8 $\wedge\, (pos \geq n \longrightarrow$
9 $String(p, first, next, prev, len, data, l_{PRE}, s[0{:}n{-}1])$
10 $\wedge\, {}^0alloc = alloc$
11 $\wedge\, Pres([\,], \{first, next, prev, len, data\})$
12 $\wedge\, res = \texttt{ERROR\_OUT\_OF\_BOUNDS})$

Note that in the postcondition we do not consider the case of function termination with the code `ERROR_OUT_OF_MEMORY`. Although the recent version of the Hoare Logic environment supports machinery for the memory consumptions computation, it was decided not to reason about this on the Hoare logic level in the frame of the Academic system. The proof of these

properties is pushed down through the stack and is supposed to be shown on the C0 semantics level.

Extensions for Guards

The length of the string's content specification is upper bounded by the `MAX_NAT` constant:

GUARD_EXT_stringInsertChar

1 $|s| \leq$ `MAX_NAT`

5.5.3 Loop Invariants

First Loop

In the first loop of the function (lines 17-20 of listing 5.4) we scan the doubly linked list of blocks in order to find the block in which a character will be inserted. The invariant for this loop is analogous to the invariant for the first loop of the `stringDeleteChar` function:

INV_1_stringInsertChar

1 $(\exists hd, tl.\ String(p, first, next, prev, len, data, hd \odot cb \odot tl, s)$
2 $\quad \wedge\ ^0pos = pos + \sum_{0 \leq i < |hd|} len(hd[i])$
3 $\quad \wedge\ \mathcal{S}(hd) \cup \{cb\} \cup \mathcal{S}(tl) \subseteq \mathcal{S}(l_{PRE}))$
4 $\wedge\ cb \neq Null$
5 $\wedge\ ^0alloc = alloc$
6 $\wedge\ Pres([\,], \{first, next, prev, len, data\})$
7 $\wedge\ res =$ `NO_ERROR`

The only modification is that we additionally state the constancy of the list *alloc* (line 5).

Second Loop

The purpose of the second loop which corresponds to lines 29-32 of the implementation is to copy the second half of the characters from the current block into the new block. Hence, in the invariant for this loop we need to show the properties of this new block:

1. It is independent of the doubly linked list relations, i.e. from the *next* and *prev* heap functions.

2. Its data prefix of the length i equals to the current block's portion of the same length starting at the position `HALF_BLOCK_SIZE`.

Additionally, we have to claim that the stated in $INV\_1\_stringInsertChar$ string relation is preserved during this loop.

Formally:

INV_2_stringInsertChar

1 $(\exists hd, tl.\ String(p, first, next, prev, len, data, hd \odot cb \odot tl, s)$
2 $\quad \wedge\ {}^0pos = pos + \sum_{0 \le i < |hd|} len(hd[i])$
3 $\quad \wedge\ \mathcal{S}(hd) \cup \{cb\} \cup \mathcal{S}(tl) \subseteq \mathcal{S}(l_{PRE}))$
4 $\wedge\ \mathcal{S}({}^0alloc) \subseteq \mathcal{S}(alloc)$
5 $\wedge\ nb \neq Null\ \wedge\ nb \in alloc$
6 $\wedge\ next(nb) = Null\ \wedge\ prev(nb) = Null$
7 $\wedge\ data(nb)[0\!:\!i-1] = data(cb)[\texttt{HALF\_BLOCK\_SIZE}\!:\!\texttt{HALF\_BLOCK\_SIZE}+i-1]$
8 $\wedge\ Pres([\,], \{first\})\ \wedge\ Pres(nb, \{next, prev, len, data\})$
9 $\wedge\ res = \texttt{NO\_ERROR}$

Thus, lines 1-3 are inherited from the first loop's invariant. Lines 5-6 correspond to the first of the two above desirable properties of the new block. Line 7 formalizes the second property, respectively. Line 8 reflects the updates of the heap-functions.

Third Loop

In the third loop which corresponds to the lines 51-54 of listing 5.4 we shift all the characters inside the current block, which lay right to the character in the position *pos* including itself one position right. This is performed in order to free a slot for the character insertion. Let us suppose the following shorthands for string intervals (the notation is the same as in the invariant for the first loop):

$$\begin{aligned} a &= \sum_{0 \le i < |hd|} len(hd[i]), \\ b &= {}^0pos - \sum_{0 \le i < |hd|} len(hd[i]), \\ c &= i - {}^0pos + 1 + \sum_{0 \le i < |hd|} len(hd[i]), \\ d &= len(cb) - i - 1, \\ e &= \sum_{0 \le i < |tl|} len(tl[i]). \end{aligned}$$

Figure 5.4 depicts the slicing of the string with respect to these intervals. The invariant for the third loop is as follows:

INV_3_stringInsertChar

1 $(\exists hd, tl, a, b, c, d, e.\ String(p, first, next, prev, len, data, hd \odot cb \odot tl,$
2 $\qquad\qquad s[0\!:\!a+b+c-1] \odot s[a+b+c-1\!:\!n-1])$
3 $\quad \wedge\ s[0\!:\!a-1] = \bigodot_{0 \le i < |hd|} data(hd[i])[0\!:\!len(hd[i])-1]$
4 $\quad \wedge\ s[a\!:\!a+b+c-1] \odot s[a+b+c-1\!:\!a+b+c+d-1] = data(cb)[0\!:\!len(cb)-1]$
5 $\quad \wedge\ s[a+b+c+d-1\!:\!n-1] = \bigodot_{0 \le i < |tl|} data(tl[i])[0 : len(tl[i]) - 1]$
6 $\quad \wedge\ \mathcal{S}(hd) \cup \{cb\} \subseteq \mathcal{S}(l_{PRE})$
7 $\quad \wedge\ (\mathcal{S}(tl) \setminus \mathcal{S}(l_{PRE})) \cap \mathcal{S}({}^0alloc) = \varnothing$

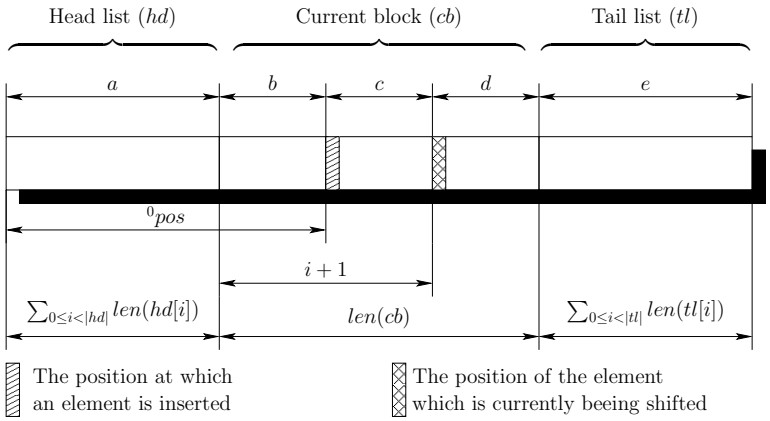

Figure 5.4: String slicing in the invariant for the third loop of the function `stringInsertChar`

```
8  ∧ (S(tl) \ S(l_PRE)) ⊆ S(alloc)
9  ∧ Pres(hd⊙cb⊙tl, {next, prev, len, data})
10 ∧ S(0alloc) ⊆ S(alloc)
11 ∧ Pres([ ], {first})
12 ∧ res = NO_ERROR
```

We state in lines 1-2 the existence of the string relation whose structure is specified by the abstract list $hd \odot cb \odot tl$. The content of the string is specified by

$$s[0 : a + b + c - 1] \odot s[a + b + c - 1 : n - 1].$$

This list corresponds to the content s of the initial string where the character in the position $i + \sum_{0 \leq i < |hd|} len(hd[i])$ is duplicated one position right. This duplication is caused by the shifting of characters. Clearly the equality $a + b + c - 1 = i + \sum_{0 \leq i < |hd|} len(hd[i])$ justifies the concatenation above.

The conditions in lines 3-5 state in a way similar to the function `stringDeleteChar` the equality between the implementations and specification of string intervals. The conditions in lines 7-8 preserve the property that the newly created block of the string in which a character has to be inserted, if there is such, is allocated. The separation properties are given in lines 9 and 11.

5.5.4 Ranking Functions

First Loop

During the first loop the position *pos* is decremented (line 19 of listing 5.4):

RANK_1_stringInsertChar

```
1 pos
```

Second Loop

With each iteration of the second loop we increment the counter i. Due to the loop exit condition in line 30 of the function's implementation and the definitions of the constants, the ranking functions looks as follows:

RANK_2_stringInsertChar

```
1 HALF_BLOCK_SIZE − i
```

Third Loop

Since the counter i is decremented during the loop it turns out that is the good candidate for the ranking function:

RANK_3_stringInsertChar

```
1 i
```

5.6 Function `subString`

This function is intended to yield a substring of a certain length started at a certain position of the given string. The specified signature of the function is:

```
string subString(string p1,
                 unsigned int start,
                 unsigned int offset)
```

where `p` is the pointer to the string manager, `start` is the position in the string beginning from which `offset` characters must be copied in order to obtain the substring. The return value of the successful execution of the function is the pointer to the substring. Otherwise `NULL` is returned.

5.6.1 Implementation

The elementary analysis of the indented function's behavior shows that, in general, the substring to be extracted may consist of several blocks. Moreover, the substring may start and end at some positions *inside* the blocks of the string. Motivated by these facts, the idea behind the algorithm is as follows:

1. Find the block in the string which corresponds to the position `start`. Create the new empty string to store the substring. Create the new empty block, and depending on the value of `offset` perform step 2 or 3.

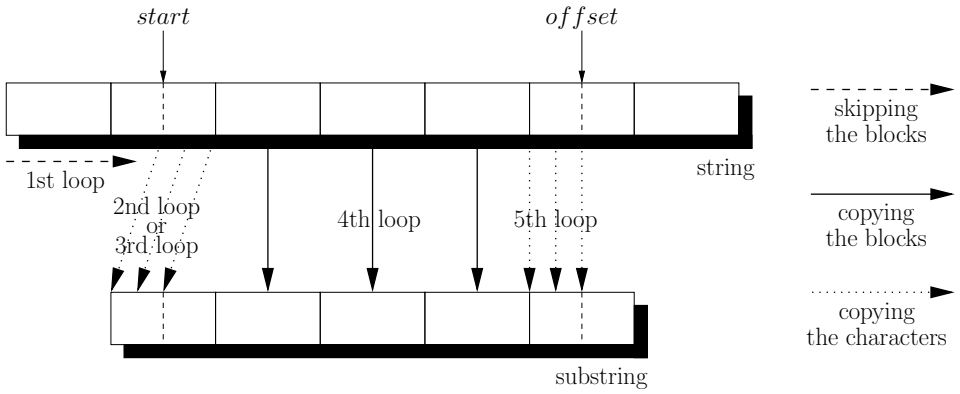

Figure 5.5: The idea behind the low-level algorithm for the function `subString` which works directly with the block structure of the string

2. In case `start` + `offset` lies within the current block of the string, copy `offset` characters from the current block starting from the position `start` (its local equivalent) into the new block. Insert the new block into the substring. Clearly, this steps ends the current branch of the algorithm.

3. In case `start` + `offset` corresponds to some further block of the string, we copy the all the data from the current block starting from the position `pos` into the new block. Insert the new block into the substring.

4. Find the block which corresponds to the position `start` + `offset`, and insert the processed during this traverse blocks into the substring (not including the block which stores the character at the position `start` + `offset` itself).

5. The current block in the string is the block found at step 4. We create the new block and copy the data from the current block up to the local equivalent of the position `start` + `offset` into the new block. Finally, we insert the new block into the string.

The steps of the algorithm are depicted in Figure 5.5.
The C0 implementation of the algorithm is as follows:

Listing 5.5: Extraction of a substring from the string

```
1 string subString(string p1, unsigned int start, unsigned int
    offset)
2 {
3   unsigned int         i,         /* counter */
4                        size;      /* length of the string */
5   string               p2;        /* substring to be returned */
6   struct string_block *cb1, *cb2, /* current blocks in strings */
```

```
7                       *nb;        /* new temporary block */
8   int                  res;       /* return-result for calls */
9
10  size = stringLength(str);
11  res  = NO_ERROR;
12
13  /* Check whether the length interval of the substring */
14  /* is contained inside the length interval of the string */
15  if (0u == offset && start < size
16   || 0u < offset && start + offset <= size) {
17
18    /* Find the block which corresponds to start */
19    cb1 = p1->first;
20    while (cb1->len <= start) {
21      start = start - cb1->len; cb1 = cb1->next;
22    }
23
24    /* Create the empty substring */
25    p2 = stringT();
26
27    if (offset != 0u) {
28
29      /* Create new empty block */
30      nb = new (struct string_block);
31      if (nb != NULL) {
32        if (start + offset <= cb1->len) {
33
34          /* Case 1: offset is within the current block */
35          nb->len = offset; i = 0u;
36
37          /* Copy data from the current block to the new block */
38          while (i < nb->len) {
39            nb->data[i] = cb1->data[start + i]; i = i + 1u;
40          }
41          p2->first = nb;
42
43        } else {
44
45          /* Case 2: offset lies in a further block */
46          nb->len = cb1->len - start; i = 0u;
47
48          /* Copy data from the current block to the new block */
49          while (i < nb->len) {
50            nb->data[i] = cb1->data[start + i]; i = i + 1u;
51          }
52          p2->first = nb;
53
54          offset = offset - nb->len;
55          cb2    = s->first;
56          cb1    = cb1->next;
57
58          /* Find the block which corresponds to the offset */
59          /* and insert traversed blocks into the string */
60          while (cb1->len < offset) {
```

```
61              nb = new (struct string_block);
62              if (nb != NULL) {
63                nb->len = cb1->len; nb->data = cb1->data;
64                res     = dlist_block_InsertAfter(cb2, nb);
65
66                if (res == NO_ERROR) {
67
68                  /* Step in the string */
69                  offset = offset - cb1->len; cb1 = cb1->next;
70
71                  /* Step in the substring */
72                  cb2 = cb2->next;
73                } else {
74                  p2 = NULL;
75                }
76              } else {
77                p2   = NULL;
78              }
79            }
80
81            if (s != NULL) {
82              nb = new (struct string_block);
83              if (nb != NULL) {
84                i = 0u; nb->len = offset;
85
86                /* Copy data from the current to the new block */
87                while (i < offset) {
88                  nb->data[i] = cb1->data[i]; i = i + 1u;
89                }
90
91                /* Insert the new block into the substring */
92                res = dlist_block_InsertAfter(cb2, nb);
93                if (res != NO_ERROR) {
94                  p2 = NULL;
95                }
96              } else {
97                p2 = NULL;
98              }
99            }
100         }
101       } else {
102         p2 = NULL;
103       }
104     }
105   } else {
106     p2 = NULL;
107   }
108
109   return p2;
110 }
```

The implementation starts with the declaration of the following variables:

- General purpose counter (`i`).
- Length of the string (`size`).
- Pointer to the substring (`p2`).
- Pointers to the current block in the string (`cb1`) and the substring (`cb2`), and to the new block (`nb`).
- Return-result of the function (`res`).

After checking whether the length interval of the substring is contained inside the length interval of the string (lines 15-16) we start searching for the block which corresponds to the position `start`. This is done via the loop in lines 18-22. With each iteration of this loop we decrement `start` by the length of the current block in order to obtain, finally, the local equivalent of the start position.

As the next step, we create the empty substring `p2` and the new empty string block `nb`. In line 32 we do the case distinction whether the position `start` + `offset` lies within the current block or somewhere after it. In the first case we copy the `offset` characters from the current block into the new block (lines 34-40). We finish this branch of the implementation by inserting this block into the string (line 41).

In the second case we copy the whole suffix (i.e. the part starting from `start` up to the block's end) of the current block into the new block (lines 48-51). Similarly to the previous case we insert the new block into the beginning of the string (line 52). After that, along lines 58-79 we search for the block which contains the character at the position `start` + `offset`. During this processing we insert the traversed block directly into the substring (lines 63-64). Now it is necessary to copy the prefix of the block which locally contains the position `start` + `offset` into the substring.

We create the new block `nb` and along lines 86-89 copy the sufficient portion of the characters into it. The implementation is finished by inserting this block into the string.

5.6.2 Formal Specification

The state space over which we formulate the specification is:

$$\Sigma = \Sigma_0 \cup (p_1 : \mathbb{R}\text{ef}, start : \mathbb{N}, offset : \mathbb{N}, i : \mathbb{N}, size : \mathbb{N}, p_2 : \mathbb{R}\text{ef}, \\ cb_1 : \mathbb{R}\text{ef}, cb_2 : \mathbb{R}\text{ef}, nb : \mathbb{R}\text{ef}, res : \mathbb{R}\text{ef}).$$

Precondition

The precondition is actually the same as in the `stringInsertChar` function:

PRE_subString

1 $String(p, first, next, prev, len, data, l_{PRE}, s[0{:}n{-}1]) \wedge \mathcal{S}(l_{PRE}) \cup \{p\} \subseteq \mathcal{S}(alloc)$

Postcondition

In the postcondition we branch on the admissibility of input parameters *start* and $offset$ — we check whether $[start : start + offset - 1] \subseteq [0 : n-1]$. In case this condition is satisfied we state in line 2 below the existence of the string relation for the string pointed to by res whose content is specified by $s[start : start + offset - 1])$ and the structure by l. In lines 3 and 4 we state that the resulting list of references l as well as the pointer res are allocated, that the allocation list grows, and that l and res are not contained in $\mathcal{S}({}^0alloc)$, respectively. Line 5 shows the separation properties. In lines 7-9 we handle the unsuccessful termination — the allocation list in this case stays unchanged, the pointer to the substring is $Null$, and the heap-functions are preserved.

Formally the postcondition is:

POST_subString

1 $(offset = 0 \wedge start < n \vee offset > 0 \wedge start + offset \leq n \longrightarrow$
2 $(\exists l.\ String(res, first, next, prev, len, data, l, s[start{:}start{+}offset{-}1])$
3 $\wedge\ \mathcal{S}(l) \cup \{res\} \subseteq \mathcal{S}(alloc)$
4 $\wedge\ (\mathcal{S}(l) \cup \{res\}) \cap \mathcal{S}({}^0alloc) = \varnothing)$
5 $\wedge\ Pres(res, \{first\})\ wedge\ Pres(l, \{next, prev, len, data\}))$
6 $\wedge \mathcal{S}({}^0alloc) \subseteq \mathcal{S}(alloc)$
7 $\wedge\ ((offset \neq 0 \vee start \geq n) \wedge (offset \leq 0 \vee start + offset > n) \longrightarrow$
8 ${}^0alloc = alloc$
9 $\wedge\ Pres([\,], \{first, next, prev, len, data\})$
10 $\wedge\ res = Null)$

Extensions for Guards

In addition to the string's content specification, we bound the end position of the supposed substring by `MAX_NAT` from the upper:

GUARD_EXT_subString

1 $|s_1| \leq$ `MAX_NAT` $\wedge\ start + offset \leq$ `MAX_NAT`

5.6.3 Loop Invariants

First Loop

In the first loop of the function (lines 20-22) we search for the string's block which corresponds to the position *start*. This operation is fairly similar to the first loop of the function `stringDeleteChar`. Formally the invariant is:

INV_1_subString

$(\exists hd, tl\ .\ String(p_1, first, next, prev, len, data, hd \odot cb_1 \odot tl, s_1)$
$\quad \wedge\ {}^0start = start + \sum_{0 \le i < |hd|} len(hd[i])$
$\quad \wedge\ \mathcal{S}(hd) \cup \mathcal{S}(tl) \cup \{cb_1\} \subseteq \mathcal{S}(l_{PRE}))$
$\wedge\ cb_1 \neq Null$
$\wedge\ \mathcal{S}(l_{PRE}) \cup \{p_1\} \subseteq \mathcal{S}(alloc)$
$\wedge\ Pres([\,], \{first, next, prev, len, data\})$
$\wedge\ res = \texttt{NO\_ERROR}$

Second and Third Loops

The structure and the intended behavior of the second (lines 38-40) and the third (lines 49-51) loops are similar. In these loops we copy the portion of the information from the current block of the string into the newly created block. The only difference between the loops is the start and end positions of that portion. Motivated by this, we specify below the common part *INV_2+3_subString* of the invariants for these loops, and then extend it with the specific for each loop assertions.

In these invariants we have to reflect basically the following facts:

1. The string's structure is decomposed into three parts: head list, current block, and tail list. The equations from the invariant for the first loop hold with respect to the partition.

2. There is the relation for the substring, such that the string's structure and its content are empty lists.

3. There is a non-*Null* new block, which is not involved in the doubly linked list relation. The data prefix of the length i (number of iterations so far) in this block equals the portion of the characters from the current block of the length i, starting from the position *start*.

INV_2+3_subString

$(\exists hd, tl\ .\ String(p_1, first, next, prev, len, data, hd \odot cb_1 \odot tl, s_1)$
$\quad \wedge\ {}^0start = start + \sum_{0 \le i < |hd|} len(hd[i])$
$\quad \wedge\ \mathcal{S}(hd) \cup \mathcal{S}(tl) \cup \{cb_1\} \subseteq \mathcal{S}(l_{PRE}))$
$\wedge\ cb_1 \neq Null$
$\wedge\ String(p_2, first, next, prev, len, data, [\,], [\,])$
$\wedge\ p_1 \neq p_2$
$\wedge\ nb \neq Null\ \wedge\ nb \notin \mathcal{S}(l_{PRE})$
$\wedge\ next(nb) = Null\ \wedge\ prev(nb) = Null$
$\wedge\ data(nb)[0\!:\!i\!-\!1] = data(cb_1)[start\!:\!start\!+\!i\!-\!1]$
$\wedge\ \{p_2, nb\} \subseteq \mathcal{S}(alloc)\ \wedge\ \{p_2, nb\} \cap (S)({}^0alloc) = \varnothing$
$\wedge\ \mathcal{S}(l_{PRE}) \cup \{p_1\} \subseteq \mathcal{S}(alloc)$

12 $\wedge\, \mathcal{S}({}^0alloc) \subseteq \mathcal{S}(alloc)$
13 $\wedge\, Pres(p_2, \{first\}) \;\wedge\; Pres(nb, \{next, prev, len, data\})$
14 $\wedge\, res = \texttt{NO\_ERROR}$

Thus, lines 1-4 formally represent the point 1 above. In line 5 we state the empty string relation for the substring. It is crucial the the pointers to the managers of the string (p_1) and the substring (p_2) differ (line 6).

Further, in lines 7-9 we formalize the status of the newly created block nb. Clearly, it has the value different from $Null$, and is disjoint with the string structure list l_{PRE} (line 7). Since this block is not inserted in the substring yet, it is independent from the doubly linked list relation — the heap functions $next$ and $prev$ yield the $Null$ value when applied to it (line 8). The content of the block nb is given in the line 9. It is obtained by the character copying at the line 39 of the function implementation (listing 5.5).

The conditions in lines 10-12 show the way how the pointers are allocated. Line 13 shows how the heaps are preserved. Since we have created the substring p_2, it changes the heap-function $first$ at this place. Creation of the new block nb may cause all the other heap-functions to change.

Concerning the invariant for the second loop, we add to the assertions above the fact that the length of the new block equals $offset$. This corresponds to the line 35 of the function implementation listing.

INV_2_subString

---

1 $INV\_2{+}3\_subString \;\wedge\; len(nb) = offset$

Consider the third loop's invariant. Since we copy the whole suffix of the current block starting from the position $start$ in this case, the length of the new block is, simply, $start$ positions less than the length of the block cb_1.

INV_3_subString

---

1 $INV\_2{+}3\_subString \;\wedge\; len(nb) = len(cb_1) - start$

Fourth Loop

During this loop which correspond to lines 60-79 of the function's listing we process the string's block structure until we find the block which stores the character at the position $start + offset$ locally. We copy the traversed through the loop blocks from the string to the substring. The following has to be stated formally in order to specify the invariant:

1. The string is partitioned into the four parts (to be defined later). The initial (global) values of the positions $start$ and $offset$ are expressed as formulas over the partition and the their local values.

2. The substring's structure is decomposed into the head list (i.e. the processed prefix) and the current block. The content of the substring

stores the processed so far characters of the string's content specification s_1 starting from the position ${}^0start$.

Thus, the formal representation of the loop invariant is as follows:

INV_4_subString

$$
\begin{array}{ll}
1 & (\exists hd_1, mdl, tl, hd_2 \,.\, String(p_1, first, next, prev, len, data, hd_1 \odot mdl \odot cb_1 \odot tl, s_1) \\
2 & \qquad \wedge\, {}^0start = start + \sum_{0 \le i < |hd_1|} len(hd_1[i]) \\
3 & \qquad \wedge\, \mathcal{S}(hd_1) \cup \mathcal{S}(mdl) \cup \mathcal{S}(tl) \cup \{cb_1\} \subseteq \mathcal{S}(l_{PRE}) \\
4 & \qquad \wedge\, String(p_2, first, next, prev, len, data, hd_2 \odot cb_2, \\
5 & \qquad\quad s_1[{}^0start : \sum_{0 \le i < |hd_1|} len(hd_1[i]) + \sum_{0 \le i < |mdl|} len(mdl[i]) - 1]) \\
6 & \qquad \wedge\, {}^0offset = \sum_{0 \le i < |mdl|} len(mdl[i]) - start + offset \\
7 & \qquad \wedge\, \mathcal{S}(l_{PRE}) \cap (\mathcal{S}(hd_2) \cup \{cb_2\}) = \varnothing \\
8 & \qquad \wedge\, \mathcal{S}(hd_2) \cup \{p_2, cb_2\} \subseteq \mathcal{S}(alloc) \\
9 & \qquad \wedge\, \mathcal{S}(hd_2) \cup \{p_2, cb_2\} \cap \mathcal{S}({}^0alloc) = \varnothing \\
10 & \qquad \wedge\, Pres(p_2, \{first\}) \;\wedge\; Pres(hd_2 \odot cb_2, \{next, prev, len, data\})) \\
11 & \wedge\, cb_1 \neq Null \;\wedge\; cb_2 \neq Null \\
12 & \wedge\, p_1 \neq p_2 \\
13 & \wedge\, \mathcal{S}(l_{PRE}) \cup \{p_1\} \subseteq \mathcal{S}(alloc) \\
14 & \wedge\, \mathcal{S}({}^0alloc) \subseteq \mathcal{S}(alloc) \\
15 & \wedge\, res = \texttt{NO\_ERROR}
\end{array}
$$

Line 1 claims the string relation. The structure of the string is decomposed into the four following parts:

1. The head part hd_1 stores the pointers to the string's blocks from the beginning up to the block which corresponds to the position *start* (not including this block itself).

2. The middle list *mdl*. It correspond to the part of the string starting right after the list hd_1 up to the current block. It stores the pointers to the blocks which are copied from the string to the substring during the loop.

3. The current block cb_1.

4. The tail list *tl* which contains all the other pointers to the string's blocks up to the end of the string.

Note, that additionally line 1 state the existence of the list hd_2, which is the "head" list for the substring. We bind this list by the same existential quantifier in order to have all the four lists (hd_1, *mdl*, *tl*, and hd_2) in one scope.

Skipping the assertions inherited from the previous invariants we consider line 4. Here the relation for the substring is stated. The content of the substring is specified as the part of the string's content starting at the position ${}^0start$ (i.e. the initial value of this variable). The end position of

the interval is given as the the sum of all blocks' lengths of the head and middle lists minus one. This is actually the position which corresponds to the processed prefix of the string.

Line 6 gives the equation for the initial value of the variable $offset$ in terms of its current value and the middle list. Line 10 claims the separation property which reflect the modifications in the substring, while all the other lines reflect either the allocation conditions of the assertion accumulated from the previous invariants.

Fifth Loop

In the last loop of this function (lines 87-89 of listing 5.5) we copy the characters from the current block into the new block up to the local equivalent of the position $start+offset$. Clearly, this loop preserves all the accumulated so far assertions regarding the string and the substring. In the invariant we nee to claim them, and the additional properties of the current block. Therefore we reuse the fourth loop's invariant in the following way:

INV_5_subString

1 $INV4_{subString}$
2 $\wedge\, nb \neq Null \;\wedge\; nb \in \mathcal{S}(alloc) \;\wedge\; nb \notin \mathcal{S}({}^{0}alloc)$
3 $\wedge\, next(nb) = Null \;\wedge\; prev(nb) = Null$
4 $\wedge\, len(nb) = offset$
5 $\wedge\, data(nb)[0{:}i{-}1] = data(cb_1)[0{:}i{-}1]$
6 $\wedge\, Pres(nb, \{next, prev, len, data\})$

Lines 2-3 describe the properties of the new block — it has some non-$Null$ value, is allocated, and does not belong to any doubly linked list relation with respect to the heap functions $next$ and $prev$. Line 4 gives its length — $offset$, which reflect the line 84 of the listing 5.5. Finally, we specify in line 5 the copied from the current block into the new block porting of data whose length is i. Line 6 additionally treats the separation of the new block.

5.6.4 Ranking Functions

First Loop

In the first loop the value of the variable $start$ decreases strictly:

RANK_1_subString

1 $start$

Second and Third Loops

The second and the third loops have the same termination conditions. While the variable i is incremented its value is bounded by the length of the new

block (lines 38-40 and 49-51 of listing 5.5). Therefore the ranking function is:

RANK_2+3_subString

1 $len(nb) - i$

Fourth Loop

The situation with the termination of this loop is more involved. The reason is that the only variable whose value is decremented is $offset$, but this decrement happens only in the case the tests in lines 62 and 66 of listing 5.5 give positive results. But we have to show the termination even if it is not the case. For this purpose we use the *arithmetization of boolean expressions* technique. We introduce the arithmetization function:

Definition 5.3 (Arithmetization Function) *The* ***arithmetization function***

$$\mathcal{A} : \mathbb{B} \to \mathbb{N}$$

interprets the boolean expression x as follows:

$$\mathcal{A}(x) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x = \mathsf{True} \\ 0 & \text{otherwise} \end{cases}.$$

Now we can construct the ranking function exploiting the arithmetization function which will handle the unsuccessful results of the mentioned above tests. The arithmetization function in this case will yield 1 and 0 on the consecutive loop iterations. Therefore the ranking function decreases. Since the both tests in case of the negative result lead to the branch where the assignment `p2 = NULL` is done, we can accumulate them in the condition $p_2 \neq Null$ to be arithmetized. Thus, the ranking function is:

RANK_4_subString

1 $offset + \mathcal{A}(p_2 \neq Null)$

Fifth Loop

Due to the increment of the counter i in this loop, the ranking function is as follows:

RANK_5_subString

1 $offset - i$

5.7 Function `stringEqual`

This function is supposed to compare two given strings according to the *lexicographical order*. This order, also called the *dictionary order*, reflects the way the words are placed in a dictionary: a sequence of characters $a_1a_2 \dots a_n$ appears in a dictionary before the string $b_1b_2 \dots b_n$ if and only if the first a_i which is different from b_i comes before b_i in the *alphabet*. That assumes both sequences have the same length. If it is not the case, the shorter word is extended with "blank"symbols. This "blank" symbol is considered as the minimum element of the alphabet. We will assume the standard ASCII order of symbols in the alphabet.

The signature of the function is:

```
int stringEqual(string p1, string p2)
```

where `p1` and `p2` are the pointers to the string managers of respective strings. The return codes of the function are:

- `EQUAL`, which denotes that the strings are equal.
- `LESS`, which occurs in case the first string is lexicographically smaller than the second string.
- `GREATER`, which occurs in case the first string is lexicographically greater than the second string.

5.7.1 Implementation

The algorithm consists of the following steps:

1. Determine the length of the shortest string.
2. Compare two strings character by character while the processed prefixes of the strings are equal. As soon as the first inequality of the characters is detected return the result of this comparison. If it is not the case and the strings have equal lengths, then they are lexicographically equal. Otherwise, the shortest string is lexicographically smaller.

Recall, that we are aimed at the low-level algorithm that works directly with string blocks. Motivated by this, we support three counters during the second step of the algorithm:

1. The global position counter, which counts the elements of an abstract string (string content).
2. The local position counter in current block of the first string. It is incremented modulo the length of the current block, i.e. it is nulled every time we process the last element of the block.

3. The same kind of the local counter for the second string.

Listing 5.6 shows the C0 implementation of the algorithm.

Listing 5.6: Lexicographical comparison of two strings

```
1  int stringEqual(string p1, string p2)
2  {
3    unsigned int i,                 /* global position */
4                 loc_i1, loc_i2,    /* local positions */
5                 size1, size2,      /* lengths of the srings */
6                 size;              /* the sorter string's length */
7    struct string_block *cb1, *cb2; /* current blocks */
8    bool         match;             /* "equality so far" flag */
9    int          res;               /* return-result */
10
11   size = 0u; i = 0u; loc_i1 = 0u; loc_i2 = 0u;
12   size1 = stringLength(p1); size2 = stringLength(p2);
13   cb1 = p1->first; cb2 = p2->first;
14
15   res    = EQUAL;
16   match  = true;
17   size   = size1;
18
19   /* Is the 1st string shorter? */
20   if (size1 < size2) { res = LESS; size = size1; }
21
22   /* Is the 2nd string shorter? */
23   if (size2 < size1) { res = GREATER; size = size2; }
24
25   /* Compare the strings as long as their prefixes are equal */
26   while (i < size && match) {
27
28     /* The character at position i in the 1st string is less */
29     if (cb1->data[loc_i1] < cb2->data[loc_i2]) {
30       res = LESS; match = false;
31     }
32
33     /* The character at position i in the 2nd string is less */
34     if (cb2->data[loc_i2] < cb1->data[loc_i1]) {
35       res = GREATER; match = false;
36     }
37
38     /* Global step */
39     i = i + 1u;
40
41     /* Local step in the 1st string */
42     if (loc_i1 + 1u == cb1->len) {
43       loc_i1 = 0u; cb1 = cb1->next;
44     } else {
45       loc_i1 = loc_i1 + 1u;
46     }
47
48     /* Local step in the 2nd string */
49     if (loc_i2 + 1u == cb2->len) {
```

```
50        loc_i2 = 0u; cb2 = cb2->next;
51      } else {
52        loc_i2 = loc_i2 + 1u;
53      }
54    }
55
56    return res;
57  }
```

We start with the declaration of the variables in lines 3-9 and their initialization in lines 11-17. These variables are:

- Global position in the string (`i`).
- Local position in the current blocks of the first (`loc_i1`) and the second (`loc_i2`) string.
- Length of the first (`size1`) and the second (`size2`) string.
- Length of the shortest string (`size`).
- Pointers to the current blocks in the first (`cb1`) and the second (`cb2`) string.
- The flag that denotes that the processed prefixes of the strings are equal (`match`).
- The return-result of the function (`res`).

The initial value of the result `res` is `EQUAL`. In the lines 19-23 we determine which of the strings is shorter and store this value in the variable `size`. We need this because we limit the number of the iterations of the functions loop (line 26) by `size` — we leave the loop if either we process the shortest string completely, or the characters being compared at the last iteration differ (i.e. `match` is false).

The body of the loop runs as follows. In the lines 28-31 and 33-36 we compare the characters at the positions `loc_i1` and `loc_i2` in the current blocks of the strings first for the "less than" criteria, then for the "greater than". The result of this comparisons is stored in the variable `res`. Clearly, if both tests yield the negative answers then `res` will hold the value from the previous iteration. After the comparisons are done we perform the step in the string, i.e. increment the global (line 39) and the local (lines 41-46 and 48-53) counters. The local counters are incremented modulo the length of the current block — if we process the last character in the block we set the local counter to zero and jump to the next block of the string.

5.7.2 Formal Specification

The state space over which we formulate the specification is:

$$\Sigma = \Sigma_0 \cup (p_1 : \mathbb{R}\text{ref}, p_2 : \mathbb{R}\text{ref}, i : \mathbb{N}, loc\_i_1 : \mathbb{N}, loc\_i_2 : \mathbb{N},$$
$$size_1 : \mathbb{N}, size_2 : \mathbb{N}, size : \mathbb{N}, cb_1 : \mathbb{R}\text{ref}, cb_2 : \mathbb{R}\text{ref}, match : \mathbb{B}, res : \mathbb{Z}).$$

Precondition

In the precondition we state the two string relations whose structure is given by the abstract lists of references l_1 and l_2, and the content by the abstract lists of characters s_1 and s_2. We suppose the object to be in the global context, i.e. visible in all the assertions regarding this function.

Formally:

PRE_stringEqual

$$\begin{array}{ll} 1 & String(p_1, first, next, prev, len, data, l_1, s_1[0{:}n-1]) \\ 2 & \wedge String(p_2, first, next, prev, len, data, l_2, s_2[0{:}m-1]) \end{array}$$

Postcondition

In order to specify the postcondition of the function we need to formalize the lexicographical comparison. For the characters $c_1, c_2 \in \mathbb{C}\text{har}$ we will write $c_1 <_{\text{lex}} c_2$ to denote that c_1 stands before c_2 in the alphabet, i.e. c_1 is lexicographically smaller than c_2. We extend this notion to the strings by interpreting $<_{\text{lex}}$ as the following function:

Definition 5.4 (Lexicographically Less Than) *Let $s_1[0 : n-1]$ and $s_2[0 : m-1]$ be abstract lists whose elements are of type $\mathbb{C}\text{har}$, and let $\sqcup$ be the blank symbol of the alphabet, i.e. its minimal symbol. The **lexicographically less then** operator is a function*

$$<_{\text{lex}}\colon L^*_{\mathbb{C}\text{har}} \times L^*_{\mathbb{C}\text{har}} \to \mathbb{B},$$

such that

$$s_1 <_{\text{lex}} s_2 \stackrel{\text{def}}{=} \begin{cases} \exists i \in \{0, \ldots, n\} \,.\, (s_1 \odot \sqcup^{m-n})[0{:}i-1] = s_2[0{:}i-1] & \\ \qquad \wedge\, (s_1 \odot \sqcup^{m-n})[i] <_{\text{lex}} s_2[i] & \text{if } n \leq m \\ \exists i \in \{0, \ldots, m\} \,.\, s_1[0{:}i-1] = (s_2 \odot \sqcup^{n-m})[0{:}i-1] & \\ \qquad \wedge\, s_1[i] <_{\text{lex}} (s_2 \odot \sqcup^{n-m})[i] & \text{otherwise} \end{cases}.$$

This definition straightforwardly matches the idea pointed in the beginning of the section — we extend the shorter string with the sequence of blank symbols and require the existence of the position inside the strings

such that the characters differ in this position, but the prefixes up to this position equals.

Thus, the postcondition states the three possible outcomes of the lexicographical comparison of the strings:

POST_stringEqual

$$
\begin{aligned}
&(s_1 <_{\text{lex}} s_2 \longrightarrow res = \texttt{LESS}) \\
\wedge&(s_2 <_{\text{lex}} s_1 \longrightarrow res = \texttt{GREATER}) \\
\wedge&(s_1 = s_2 \longrightarrow res = \texttt{EQUAL})
\end{aligned}
$$

Extensions for Guards

The lengths of both strings' content specifications are upper bounded by the `MAX_NAT` constant:

GUARD_EXT_stringEqual

$$|s_1| \leq \texttt{MAX\_NAT} \wedge |s_2| \leq \texttt{MAX\_NAT}$$

5.7.3 Loop Invariant

In the invariant for the function's loop (lines 26-54) we need to reflect, basically, two facts:

1. We process the string according to its block structure, and the string, therefore, can be partitioned into the "head" list, current block, and the "tail" list.

2. At each iteration of the loop the result `res` stores one of the possible values, namely, `EQUAL`, `LESS`, and `GREATER`, depending on the current value of the flag *match*.

Formally, the loop invariant is as follows:

INV_stringEqual

$$
\begin{aligned}
&\forall k \in \{1,2\}. \\
&(cb_k \neq Null \longrightarrow \exists hd_k, tl_k. String(p_k, first, next, prev, len, data, hd_k \odot cb_k \odot tl_k, s_k) \\
&\qquad \wedge i = \sum_{0 \leq i < |hd_k|} len(hd_k[i]) + loc\_i_k) \\
\wedge&(cb_k = Null \longrightarrow i = |s_k|) \\
\wedge&(match \longrightarrow s_1[0{:}i{-}1] = s_2[0{:}i{-}1] \\
&\qquad \wedge (|s_1| = |s_2| \longrightarrow res = \texttt{EQUAL}) \\
&\qquad \wedge (|s_1| < |s_2| \longrightarrow res = \texttt{LESS}) \\
&\qquad \wedge (|s_2| < |s_1| \longrightarrow res = \texttt{GREATER})) \\
\wedge&(\neg match \longrightarrow s_1[0{:}i{-}2] = s_2[0{:}i{-}2] \\
&\qquad \wedge s_1[i-1] \neq s_2[i-1] \\
&\qquad \wedge (s_1[i-1] < s_2[i-1] \longrightarrow res = \texttt{LESS}) \\
&\qquad \wedge (s_2[i-1] < s_1[i-1] \longrightarrow res = \texttt{GREATER}))
\end{aligned}
$$

We need to state the status of both string in the invariant and the assertions for these strings have similar parts. Hence, we use the universal quantification over the indicies $k \in \{1, 2\}$ of the variables which correspond to the first and the seconds strings, respectively (line 1 of the invariant above). Thus, in lines 2-5 we consider the two possible values of the current block in the first and the second strings.

If the pointer to the current block is not $Null$, then it is located inside the string between the head list and the tail list (line 2). The reason for this partition and its details are the same as for the function `stringDeleteChar` (see section 5.4.3). Line 3 states that the current value of the global position counter i is nothing but the sum of the block lengths of the head list and the local position counter in the current block.

Conversely, the $Null$-value of the pointer to the current block indicates that the string has been processed completely. Therefore, the global position counter equals in this case to the length of this string's content specification (line 4).

Further, in lines 5-12 we specify two outcomes depending on the current comparison result *match*.

Firstly, we consider the case when *match* holds. The string's prefixes of the length i are, therefore, equal (line 5). According to the definition 5.4 the result of the lexicographical comparison (here — its intermediate value) depends on the strings' lengths in this case. We specify the three possible outcomes in lines 4-8.

Secondly, we specify the case when *match* is false. Recall listing 5.6. We exit the loop as soon as *math* does not hold. Therefore, we can conclude that if *match* is false at the current iteration of the loop, then it was true at the previous step. This implies that the string prefixes of the length $i-1$ are equal (line 10). The result of the lexicographical comparison depends then on the characters in the position [2] $i - 1$. Lines 11-12 specify the possible outcomes.

5.7.4 Ranking Function

During the loop the global counter i is incremented until it reaches the value $size$. Therefore the ranking function is:

RANK_stringEqual

```
1 size − i
```

5.8 Function `stringFind`

The functions searches for the first occurrence of the one string in another, starting at a certain position. The signature is:

[2] Recall, that we count characters in the string starting from 0.

```
int stringFind(string p1, string p2, unsigned int pos)
```

where `p1` and `p2` are the pointers to the string managers of first and second strings. We will refer to these strings as *string* and *substring*, respectively. The starting position for the search in the string is given by the parameter `pos`. The function terminates with the following return-values:

- `ERROR_INVALID` indicates that at least one of the strings is empty and we are not able to perform the search in this case.
- `ERROR_OUT_OF_BOUNDS` occurs in case the sum of the `pos` and the length of the substring is greater than the size of the string. In this case we are not able to match the substring with the sufficient portion of the string completely.
- `ERROR_NOT_FOUND` is returned if the substring was not found.
- some non-negative value x is returned in case the substring was found at position x.

5.8.1 Implementation

There are different algorithms for the string matching. The basic algorithm runs in quadratic time and does not use additional space. Those algorithms, that have asymptotically smaller time bounds involve additional data structures to store the result of the string preprocessing (see, for example, [CLR03] section 32). Since we are not interested in handling additional data structures, and therefore, formalizing them and accumulating the lemmata about their properties, which clearly increase the verification time, we implement the basic algorithm.

The steps of the algorithm are as follows:

1. Test for the first two return-codes. If these tests fail, find the block which corresponds to the position `pos`.
2. Start matching the string from the position `pos` and the substring starting from the zero position character by character. If the substring matches the string, return the starting position of the match.
3. Otherwise, increment the starting position of the match, and repeat step 2.

Logically steps 2 and 3 are implemented via the nested loop. In the outer loop we increment the starting position of the match, and in the inner loop we perform the consecutive matching of the strings.

Analogically to the function `stringEqual` we distinguish between global and local positions. The global positions count the characters in the abstract

strings. The local counters store the positions of the characters we are looking at in the current blocks.

Listing 5.7 depicts the C0 implementation of the function.

Listing 5.7: Search for the substring

```
1  int stringFind(string p1, string p2, unsigned int pos)
2  {
3    unsigned int i, j,              /* global counters */
4                 loc_i, loc_j,      /* local counters */
5                 rb_i,              /* rollback position */
6                 size1, size2;      /* lengths of the strings */
7    struct string_block *cb1, *cb2, /* current blocks */
8                        *rb;        /* rollback block */
9    bool         match;             /* "equality so far" flag */
10   int          res;               /* return-result */
11
12   res   = ERROR_NOT_FOUND;
13   size1 = stringLength(p1); size2 = stringLength(p2);
14
15   if (size1 == 0u || size2 == 0u) {
16     res = ERROR_INVALID;
17   } else {
18     if (size1 < pos + size2) {
19       res = ERROR_OUT_OF_BOUNDS;
20     } else {
21
22       i = pos; loc_i = pos; cb1 = p1->first;
23
24       /* Find the block which corresponds to pos */
25       while (cb1->len <= loc_i) {
26         loc_i = loc_i - cb1->len; cb1 = cb1->next;
27       }
28
29       /* Save the rolback block and local position */
30       rb = cb1; rb_i = loc_i;
31
32       /* Loop over the string */
33       while (i + size2 <= size1 && res == ERROR_NOT_FOUND) {
34
35         cb2 = p2->first; j = 0u; loc_j = 0u;
36         match = true;
37
38         /* Loop over the substring */
39         while(j < size2 && match) {
40
41           if (cb1->data[loc_i] != cb2->data[loc_j]) {
42             match = false;
43           } else {
44
45             /* Global step in the substring */
46             j = j + 1u;
47
48             /* Local step in the substring */
```

```
49              if (loc_j + 1u == cb2->len) {
50                loc_j = 0u; cb2 = cb2->next;
51              } else {
52                loc_j = loc_j + 1u;
53              }
54
55              /* Local step in the string */
56              if (loc_i + 1u == cb1->len) {
57                loc_i = 0u; cb1 = cb1->next;
58              } else {
59                loc_i = loc_i + 1u;
60              }
61            }
62          }
63
64          /* Rollback in case of mismatch */
65          if (match) {
66            res = int(i);
67          } else {
68
69            /* Rollback and start matching again */
70            if (rb_i + 1u == rb->len) {
71              rb_i = 0u; rb = rb->next;
72            } else {
73              rb_i = rb_i + 1u;
74            }
75            cb1 = rb; loc_i  = rb_i;
76
77            /* Global step in the string */
78            i = i + 1u;
79          }
80        }
81      }
82    }
83
84    return res;
85  }
```

The implementation starts with the declaration of the variables in lines 3-10 and consecutive initialization of some of them. These variables are:

- Global position in the string (`i`) and in the substring (`j`)
- Local position in the current block of the first (`loc_i`) and the second (`loc_j`) string.
- Local position in the rollback block (the description what the rollback is see below) of the string (`rb_i`). We start the repeated matching from the position `rb_i` + 1 modulo the length of the rollback block in case of the mismatch.
- Length of the first (`size1`) and the second (`size2`) string.

- Pointers to the current block of the first (`cb1`) and the second (`cb2`) string.
- The flag `match` which denotes that the processed prefix of the substrings is contained in the string.
- The return-result of the function (`res`).

In lines 15-16 we test whether the length of the string or of the substring is zero. In case it is, we signal this with the `ERROR_INVALID` return-value. The lines 18-19 perform the test to find out whether the parameter `pos` is given sufficiently to perform the matching completely. If it is not the case, we return the `ERROR_OUT_OF_BOUNDS` code.

As the next step we search the string in order to find the block which stores the character at the position `pos` (lines 24-27). This is done in the same manner as it was done in the previously described functions (see `stringDeleteChar`, section 5.4 for example). As soon as we are done with this, we save the pointer to current block in the string to the rollback block variable `rb`, and the local position `loc_i` to the local rollback position `rb_i` (line 30). This will allow us to make the rollback to the sufficient position in case of the mismatch in the future.

In line 33 we start the loop over the string. The substring is processed from the very beginning at each iteration of this loop, therefore the corresponding parameters of the substring are initialized in lines 35-36. These initialisations are followed by the loop over the substring (line 39). We test the corresponding characters, namely the symbol at the position `loc_i` in the block `cb1` and the character at the position `loc_j` in the block `cb2`, for the inequality. If they differ, we set `match` to false, which implies that we exit from the inner loop. Otherwise, we need to make a step in the string and in the substring. We increment the global position in the substring (line 46) and the local positions in the substring (lines 48-53) and in the string (lines 55-60). This increments are done in the same manner as for the function `stringEqual` (see section 5.7.1 for details).

After we exit the inner loop we need to determine whether we succeeded in the matching and need to return from the function, or the previous match was unsuccessful and we need to repeat it starting from the next position in the string. This is done along the lines 64-79. If `match` holds, we store the value of the position in the string (`i`) converted to the integer type in the result variable `res` (line 66). This type casting is necessary since we decided that: i) the result of function `res` will share the negative error codes and the positive non-error values, and ii) all the lengths and positions in all the functions will be represented as unsigned integers. If `match` is false we do the rollback to the position which is one character after the previously saved rollback position (lines 69-75). The step to the next character in the string (line 78) finishes the outer loop of the function.

5.8.2 Formal Specification

The state space over which we formulate the specification is:

$$\Sigma = \Sigma_0 \cup (p_1 : \mathbb{R}\text{ef}, p_2 : \mathbb{R}\text{ef}, pos : \mathbb{N}, i : \mathbb{N}, j : \mathbb{N}, loc\_i : \mathbb{N}, loc\_j : \mathbb{N}, rb\_i : \mathbb{N},$$
$$size_1 : \mathbb{N}, size_2 : \mathbb{N}, cb_1 : \mathbb{R}\text{ef}, cb_2 : \mathbb{R}\text{ef}, rb : \mathbb{R}\text{ef}, match : \mathbb{B}, res : \mathbb{Z}).$$

Precondition

The precondition is the same as in `stringEqual` function:

PRE_stringFind

$$1 \quad String(p_1, first, next, prev, len, data, l_1, s_1[0{:}n-1])$$
$$2 \quad \wedge String(p_2, first, next, prev, len, data, l_2, s_2[0{:}m-1])$$

Postcondition

The postcondition reflects the specified above outcomes of the function rather straightforwardly. It follows that the precondition of this function have many nested branches and expressing them via the implications will look clumsy. In order to avoid this, we define the `if...then...else...` syntactic abbreviation:

Convention 5.5 *Let x, y, and z be logical expressions. Then*

$$\texttt{if } x \texttt{ then } y \texttt{ else } z \stackrel{\text{def}}{=} (x \longrightarrow y) \wedge (\neg x \longrightarrow z).$$

With this notational convention the postcondition looks as follows:

POST_stringFind

```
1 if (n = 0 ∨ m = 0) then res = ERROR_INVALID
2 else if (n < pos + m) then res = ERROR_OUT_OF_BOUNDS
3 else if (∃k . pos ≤ k ∧ s1[k:m − 1] = s2) then
4  (pos ≤ res ∧ s1[res:m − 1] = s2 ∧ ∀k . pos ≤ k < res ⟶ s1[k:m − 1] ≠ s2)
5  else res = ERROR_NOT_FOUND
```

In line 1 we specify the case when one of the strings is empty. In line 2 we handle the case when the passed parameter *pos* does not fit the lengths of the string (n) and the substring (m). Lines 3-4 specify the case when the substring is found starting from some position $k \geq pos$. The result res stores then position at which the substring is found, and additionally we claim that the substring could not be found at any position before, in the range $\{pos, \ldots, res - 1\}$. Line 5 finishes the specification by stating the remaining outcome `ERROR_NOT_FOUND`.

Extensions for Guards

Recall that the return-result variable of this function is of integer type and it can store negative error codes as well as non-negative positions. Hence, we have to bound the strings' sizes by the `MAX_INT` constant in this case. In addition to that we bound the end position of the substring as well:

GUARD_EXT_stringFind

1 $|s_1| \leq \texttt{MAX\_INT} \wedge pos + |s_2| \leq \texttt{MAX\_INT}$

5.8.3 Loop Invariants

First Loop

In the first loop of the function (lines 25-27) we are searching for the block in the string which corresponds to the position *pos*. The invariant for this loop reflects three facts:

1. The string is partitioned according to the "head — current block — tail" scheme (analogically to the `stringDeleteChar` function, for example).

2. Position *pos* is the sum of processed up to the current block block lengths and the local position in this block.

3. The the string relation for the substring is preserved during the loop.

Formally:

INV_1_stringFind

1 $(\exists hd, tl \;.\; String(p_1, first, next, prev, len, data, hd \odot cb_1 \odot tl, s_1)$
2 $\qquad \wedge pos = \sum_{0 \leq i < |hd|} len(hd[i]) + loc\_i)$
3 $\wedge\; (\exists l \;.\; String(p_2, first, next, prev, len, data, l, s_2))$

Thus, line 1 states the possibility of the string partition. Line 2 reflects the fact that the global position *pos* corresponds to the local position *loc_i* in the current block, i.e. the first block after the list *hd*. Line 3 states that the substring stays unchanged during the loop.

Second (Outer) Loop

Recall, that the matching of the two strings is implemented via the nested loop. The outer loop (lines 33-80) is essentially intended to shift the starting position i of the matching in the first string. If the matching during the inner loop fails, we return to the starting position in the string shifted one step right. In order to express this idea formally we must state the following facts in the invariant:

1. The string is partitioned into the "head" and "tail" lists, such that the rollback block is located between them in the specification of the string's structure.

2. The starting position of the match i is the sum of the block lengths in the "head" list and the local rollback position.

3. The relation for the second string is preserved during the loop.

4. The substring could not start in the position range $\{pos, \ldots, i-1\}$.

5. Depending of the result of the search we either rollback, or return from the function.

Formally, the loop invariant is as follows:

INV_2_stringFind

1 $(i + size_2 \leq size_1 \longrightarrow$
2 $(\exists hd, tl \,.\, String(p_1, first, next, prev, len, data, hd \odot rb \odot tl, s_1)$
3 $\wedge\, (res = \texttt{ERROR\_NOT\_FOUND} \longrightarrow i = \sum_{0 \leq i < |hd|} len(hd[i]) + rb\_i))$
4 $\wedge\, (\exists l \,.\, String(p_2, first, next, prev, len, data, l, s_2))$
5 $\wedge\, (\forall k \,.\, pos \leq k < i \longrightarrow s_1[k{:}m{-}1] \neq s_2)$
6 $\wedge\, (res = \texttt{ERROR\_NOT\_FOUND} \longrightarrow cb_1 = rb \;\wedge\; loc\_i = rb\_i)$
7 $\wedge\, (res \neq \texttt{ERROR\_NOT\_FOUND} \longrightarrow s_1[i{:}m{-}1] = s_2 \;\wedge\; res = i)$

If the condition in line 1 holds then we are inside the loop. In this case the string's block structure is partitioned into three parts: $hd \odot rb \odot tl$. This denotes that the rollback block is inside the string (line 2). Recall the function's implementation. In case of mismatch the result variable res stores the `ERROR_NOT_FOUND` code, and exactly in this case we have to do the rollback. Line 3, therefore, provides the equation which states how the matching start position i is related to the local rollback position. Line 4 preserves the second string relation.

Line 5 straightforwardly states the fact that all the previous attempts to match the strings, i.e. starting at any position k, such that $pos \leq k < i$, failed.

Line 6 reflects the assignments at the line 75 of listing 5.7. Naturally, in case we are at the end of the outer loop and have a mismatch, we set the current block of the string to the rollback block, and we set the local position to the rollback position. Line 7 treats the opposite case, namely when we found the substring inside the string. In this case we store the position i in res.

Third (Inner) Loop

In the inner loop which corresponds to lines 39-62 we process the strings character by character in order to find out whether they match together. We

process the string starting from the position i and the substring, logically, from the very beginning. Since this processing requires the traverse of the strings' block structure, in the invariant we have to state the following:

1. Both strings are partitioned (the partition scheme is defined below), and the positions counters are expressed as equations with respect to the partitions.

2. The substring could not start in the position range $\{pos, \ldots, i-1\}$.

3. In case of match the position of the string of length j starting from the position i equals to the prefix of the substring of length j, otherwise the character at the position $i+j$ in the string differs from the character in the position j of the substring.

We state these facts formally:

INV_3_stringFind

$$
\begin{aligned}
&(j < size_2 \longrightarrow \\
&\quad (\exists hd, mdl, tl \,.\, String(p_1, first, next, prev, len, data, hd \odot mdl \odot cb_1 \odot tl, s_1) \\
&\qquad \wedge (mdl \odot cb_1)[0] = rb \\
&\qquad \wedge i = \sum_{0 \le i < |hd|} len(hd[i]) + rb\_i \\
&\qquad \wedge i + j = \sum_{0 \le i < |hd|} len(hd[i]) + \sum_{0 \le i < |mdl|} len(mdl[i]) + loc\_i) \\
&\quad \wedge (\exists hd, tl \,.\, String(p_2, first, next, prev, len, data, hd \odot cb_2 \odot tl, s_2) \\
&\qquad \wedge j = \sum_{0 \le i < |hd|} len(hd[i]) + loc\_j) \\
&\wedge (\forall k \,.\, pos \le k < i \longrightarrow s_1[k\!:\!m-1] \neq s_2) \\
&\wedge (match \longrightarrow s_1[i\!:\!j-1] = s_2[0\!:\!j-1]) \\
&\wedge (\neg match \longrightarrow s_1[i+j] \neq s_2[j])
\end{aligned}
$$

In the invariant we need to reflect that the string contains the rollback block rb and the current block cb_1 at the same time. According to the algorithm these two block initially, i.e. before the first iteration of the inner loop, coincide. During the loop execution, the position of the rollback block stays unchanged. The current block is, oppositely, shifted to the right. Therefore we slice the abstract list specifying the string's structure into the following parts:

1. Head list hd which contains the pointers to blocks before the rollback block.

2. Middle list mdl which is initially empty, and then contains the pointer to the blocks between the rollback block and the current block. It includes the rollback block.

3. Current block cb_1.

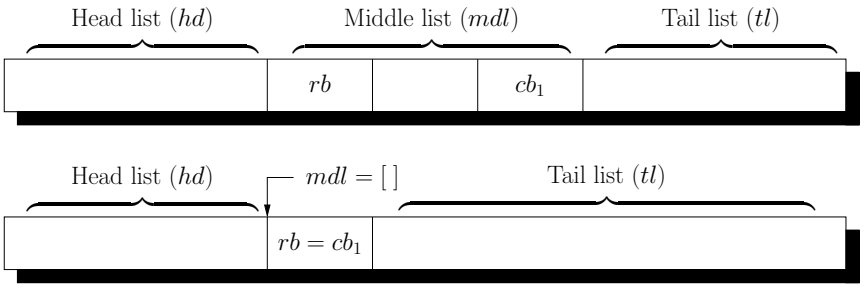

Figure 5.6: Partition of the string in the third (inner) loop of the function `stringDeleteChar` before the first iteration (below) and during the loop (above)

4. Tail list tl which contains the pointers to the blocks after the current block.

The partition of the string with respect of these intervals is depicted in Figure 5.6.

Thus, if we are inside the inner loop (i.e. the condition in line 1 of the invariant above holds), the string is partitioned into four parts: hd, mdl, cb_1, and tl (line 2). In line 3 we formally state the fact that the rollback block either coincide with the current block (in case $mdl = [\,]$), or is the first element of the middle list. Line 4 reflects, analogically to the outer loop, the value of the global position counter i (the start position of the matching). The next line is needed to connect the global counter of the matching progress j with the local position counters.

In lines 6-7 we slice the substring according to our "ordinary" scheme, i.e. into the three parts: hd, cb_2, and tl, and express the global position in the substring j in terms of processed so far blocks, and local position counter. Line 8 states the same condition as for the outer loop. Lines 9-10 branch on the $match$ — in case it holds the portion of the string from the position i up to the position $j-1$ equals the prefix of the substring of length j; in case it does not hold the characters in the current position of the comparison differ.

5.8.4 Ranking Functions

First Loop

The first loop strictly decreases the variable loc_i which we take as the ranking function:

	$RANK\_1\_stringFind$
1	loc_i

Second (Outer) Loop

In the outer the two cases are possible: either the counter i is incremented (line 78 of listing 5.7) of we assign to the variable res the non-negative value (line 66), which immediately violates the second conjunct in the loop entrance condition (line 33). The counter i is incremented until it becomes equal to $size_1 - size_2$. Thus, the ranking function is:

RANK_2_stringFind

1 $size_1 - size_2 - i$

Third (Inner) Loop

In the ranking function for the last loop we exploit the arithmetization of booleans technique in order to take into account the branch of the negative test outcome (line 41-42). For the other branch we consider the variable j which is incremented. Therefore, the ranking function is as follows:

RANK_3_stringFind

1 $size_2 - j + \mathcal{A}(match)$

Chapter 6

Proof of Verification Conditions

In this chapter we sketch the proof steps of the function `stringDeleteChar`. This function has two loops, and therefore its proof is enough complex to illustrate all basic ideas and techniques which were used during the proof of the library in the Hoare Logic environment. Firstly, we prove the partial correctness of this function. Then, we show the arguments for the termination. We do not show the proofs for the conditions generated due to the guards since they are relatively simple and do not differ much from the subgoals presented in the example from section 3.3.

6.1 Review

Recall the precondition:

$PRE\_stringDeleteChar$

$String(p, first, next, prev, len, data, l_{PRE}, s[0:n-1])$

Recall the postcondition:

$POST\_stringDeleteChar$

$$
\begin{array}{l}
(pos < n \longrightarrow \\
\quad (\exists l.\ String(p, first, next, prev, len, data, l, s[0:pos-1] \odot s[pos+1:n-1]) \\
\quad \wedge\ \mathcal{S}(l) \subseteq \mathcal{S}(l_{PRE})) \\
\quad \wedge\ res = \texttt{NO\_ERROR} \\
\quad \wedge\ Pres(p, \{first\})\ \wedge\ Pres(l_{PRE}, \{next, prev, len, data\})) \\
\wedge\ (pos \geq n \longrightarrow \\
\quad String(p, first, next, prev, len, data, l_{PRE}, s[0:n-1]) \\
\quad \wedge\ Pres([\,], \{first, next, prev, len, data\}) \\
\quad \wedge\ res = \texttt{ERROR\_OUT\_OF\_BOUNDS})
\end{array}
$$

Recall the first invariant:

INV_1_stringDeleteChar

$$
\begin{aligned}
&(\exists hd, tl.\ String(p, first, next, prev, len, data, hd \odot cb \odot tl, s) \\
&\qquad \wedge\ {}^{0}pos = pos + \sum_{0 \le i < |hd|} len(hd[i]) \\
&\qquad \wedge\ \mathcal{S}(hd) \cup \{cb\} \cup \mathcal{S}(tl) \subseteq \mathcal{S}(l_{_{PRE}})) \\
&\wedge\ cb \neq Null \\
&\wedge\ Pres([\,], \{first, next, prev, len, data\}) \\
&\wedge\ res = \texttt{NO\_ERROR}
\end{aligned}
$$

Recall the second invariant:

INV_2_stringDeleteChar

$$
\begin{aligned}
&(\exists hd, tl, a, b, c, d, e.\ String(p, first, next, prev, len, data, hd \odot cb \odot tl, \\
&\qquad\qquad s[0\!:\!a\!+\!b\!-\!1] \odot s[a\!+\!b\!+\!1\!:\!a\!+\!b\!+\!c] \odot s[a\!+\!b\!+\!c\!:\!n\!-\!1]) \\
&\qquad \wedge\ s[0\!:\!a\!-\!1] = \bigodot_{0 \le i < |hd|} data(hd[i])[0\!:\!len(hd[i])\!-\!1] \\
&\qquad \wedge\ s[a\!:\!a\!+\!b\!-\!1] \odot s[a\!+\!b\!+\!1\!:\!a\!+\!b\!+\!c] \odot s[a\!+\!b\!+\!c\!:\!a\!+\!b\!+\!c\!+\!d\!-\!1] \\
&\qquad\qquad = data(cb)[0\!:\!len(cb)\!-\!1] \\
&\qquad \wedge\ s[a\!+\!b\!+\!c\!+\!d\!:\!n\!-\!1] = \bigodot_{0 \le i < |tl|} data(tl[i])[0\!:\!len(tl[i])\!-\!1] \\
&\qquad \wedge\ \mathcal{S}(hd) \cup \{cb\} \cup \mathcal{S}(tl) \subseteq \mathcal{S}(l_{_{PRE}})) \\
&\wedge\ Pres(cb, \{data\}) \ \wedge\ Pres([\,], \{first, next, prev, len\}) \\
&\wedge\ res = \texttt{NO\_ERROR}
\end{aligned}
$$

We will refer to these assertions simply by the shorthands PRE, $POST$, INV_1, and INV_2. Since the function `stringDeleteChar` has two loops the following five verification conditions are generated according to the weakest precondition strategy (recall section 3.2.2):

1. Implication from the precondition to the first invariant.
2. Preservation of the first invariant.
3. Implication from the first invariant to the second invariant.
4. Preservation of the second invariant.
5. Implication from the second invariant to the postcondition.

We formalize these conditions in the respective section below and will refer to their formal representations by the aliases VC_1 - VC_5.

6.2 Showing the Partial Correctness

6.2.1 From the Precondition to the First Invariant

The first verification condition we want to show is:

VC_1

$$(PRE \wedge pos < n \longrightarrow INV_1[res := \texttt{NO\_ERROR}, cb := first(p)])$$
$$\wedge (PRE \wedge pos \geq n \longrightarrow POST[res := \texttt{ERROR\_OUT\_OF\_BOUNDS}])$$

We apply the substitutions and continue with showing the second conjunct of VC_1.

Consider the postcondition $POST$. Since the assumption $pos < n$ of its first implication is False it suffices to show only its second implication:

$$\wedge\, (pos \geq n \longrightarrow$$
$$String(p, first, next, prev, len, data, l_{PRE}, s[0\!:\!n\!-\!1])$$
$$\wedge\, Pres([\,], \{first, next, prev, len, data\})$$
$$\wedge\, res = \texttt{ERROR\_OUT\_OF\_BOUNDS}).$$

This case is also simple since i) we have in the assumption PRE the string relation we need to show in $POST$, and ii) the heap functions were not updated in this branch of the algorithm and, therefore, are the same as they were in the precondition.

To show the first conjunct of VC_1 we unfold the definitions of $String$ and $dList$ and obtain, after the elimination of quantifiers, the following:

$$PRE = \ldots \wedge List(first(p), next, l_{PRE}) \wedge List(q, prev, l_{PRE}^{-1}) \wedge \ldots.$$

We instantiate the existential quantifiers in INV_1 with [ ] and $l_{PRE}[1\!:\!|l_{PRE}|\!-\!1]$ respectively and after simplification obtain:

$$INV\_1 =$$
$$List(prev(first(p)), prev, [\,]) \wedge List(next(first(p)), next, l_{PRE}[1\!:\!|l_{PRE}|\!-\!1])$$
$$\wedge \{first(p)\} \cup \mathcal{S}(l_{PRE}[1\!:\!|l_{PRE}|\!-\!1]) \subseteq \mathcal{S}(l_{PRE}) \wedge first(p) \neq Null.$$

It is easy to see that from singly list relations in PRE the equality

$$prev(first(p)) = prev(l_{PRE}[0]) = prev(l_{PRE}^{-1}[|l_{PRE}|\!-\!1]) = Null$$

follows from definitions of $List$ and $Path$. Hence, this shows the first list relation in INV_1. The second list relation follows from PRE since the former is a subformula of the latter.

To show that $\{first(p)\} \cup \mathcal{S}(l_{PRE}[1 : |l_{PRE}| - 1]) \subseteq \mathcal{S}(l_{PRE})$ we exploit the fact $first(p) = l_{PRE}[0]$ from which we conclude that $first(p) \odot l_{PRE}[1 : |l_{PRE}| - 1])) = l_{PRE}$.

The last conjunct of INV_1 follows from PRE by definitions of $List$ and $Path$ — references which occur in these relations must not be $Null$. Since all conjuncted terms in the conclusion of VC_1 hold the first verification condition holds.

6.2.2 Preservation of the First Invariant

The second verification condition is:

VC_2
$INV_1 \wedge pos \geq len(cb) \longrightarrow INV_1[cb := next(cb), pos := pos - len(cb)]$

Let us refer to $INV\_1[cb := next(cb), pos := pos - len(cb)]$ after the application of substitution as to INV_1'. We eliminate the quantifiers in INV_1 and instantiate the existential quantifiers in INV_1' for head and tail lists with $cb \odot hd$ and $tl[1 : |tl| - 1]$ respectively. This instantiation reflects the traversal inside the structure specification list of the string — on each iteration of the loop we assign the next block to the current one. After arithmetic simplifications and omitting conjuncts which occur in the assumption of the implication in VC_2 we obtain:

$INV\_1' =$
$String(p, first, next, prev, len, data, hd \odot cb \odot next(cb) \odot tl[1 : |tl| - 1], s)$
$\wedge\, \mathcal{S}(hd) \cup \{cb, next(cb)\} \cup \mathcal{S}(tl[1 : |tl| - 1]) \subseteq \mathcal{S}(l_{PRE}) \wedge next(cb) \neq Null$

Unfolding the definitions in INV_1 down to the level of $Path$ one concludes that $next(cb) = tl[0]$. Therefore $hd \odot cb \odot next(cb) \odot tl[1 : |tl| - 1] = hd \odot cb \odot tl$ and the string relation in INV_1' holds. Nearly the same reasoning proves the second conjunct of INV_1'.

To show condition $next(cb) \neq Null$ recall the $Path$ definition. All the references in the path must not be $Null$. Therefore $next(cb) = tl[0]$ implies $next(cb) \neq Null$. This finishes the proof of VC_2.

6.2.3 From the First to the Second Invariant

The third verification condition we want to show is:

VC_3
$INV_1 \wedge pos < len(cb) \longrightarrow INV_2$

We instantiate the existential quantifiers in INV_2 for the head and tail lists with hd and tl, respectively. We instantiate the existential quantifiers for the string intervals exactly with the same values as it is specified in section 5.4.3. Together with the assumption ${}^0pos = pos + \sum_{0 \leq i < |hd|} len(hd[i])$ from INV_1 equations for the string intervals result in $c = 0$ in INV_2. Therefore the part $s[a + b + 1 : a + b + c]$ of string specification in INV_2 boils down to $[\,]$. With this simplification the string relation in INV_2 is equivalent to the string relation of implication assumption in VC_3 and therefore holds.

To show the conditions about the equality of implementation and specification of the string parts in INV_2 we unfold the definition of $String$ in

INV_1 obtaining:

$$\begin{aligned} INV\_1 \;=\; & \ldots \wedge s = \bigodot_{0 \le i < |hd^{-1}|} data(hd^{-1}[i])[0 : len(hd^{-1}[i]) - 1] \\ & \odot data(cb)[0 : len(cb) - 1] \\ & \odot \bigodot_{0 \le i < |tl|} data(tl[i])[0 : len(tl[i]) - 1] \wedge \ldots . \end{aligned}$$

Consider the string intervals. It turns out that

$$a = \sum_{0 \le i < |hd|} len(hd[i]) = \sum_{0 \le i < |hd^{-1}|} len(hd^{-1}[i]).$$

Hence the condition for the first string interval

$$s[0 : a - 1] = \bigodot_{0 \le i < |hd^{-1}|} data(hd^{-1}[i])[0 : len(hd^{-1}[i]) - 1]$$

follows from INV_1. Analogously to the equality

$$a + b + d = \sum_{0 \le i < |hd|} len(hd[i]) + len(cb) = \sum_{0 \le i < |hd^{-1}|} len(hd^{-1}[i]) + len(cb)$$

we are able to show

$$s[a : a + b + d - 1] = data(cb)[0 : len(cb) - 1].$$

Finally from

$$n = a + b + d + e = \sum_{0 \le i < |hd^{-1}|} len(hd^{-1}[i]) + len(cb) + \sum_{0 \le i < |tl|} len(tl[i])$$

and INV_1 the condition for the last string interval

$$s[a + b + d : n - 1] = \bigodot_{0 \le i < |tl|} data(tl[i])[0 : len(tl[i]) - 1]$$

follows.

The part of the proof which corresponds to the separation properties is:

$$\begin{aligned} & Pres([\,], \{first, next, prev, len, data\}) \longrightarrow \\ & Pres(cb, \{data\}) \wedge Pres([\,], \{first, next, prev, len\}). \end{aligned}$$

Clearly, it is easy to show since we have the constancy of the heap-functions for all the references in the assumption of the implication, and we impose restrictions on this set of references in the conclusion.

The remaining conjuncts of INV_2 are obvious since they occur in the assumption INV_1 of the implication in VC_3.

6.2.4 Preservation of the Second Invariant

The fourth verification condition to be shown is:

VC_4

$$INV_2 \wedge pos < len(cb) - 1 \longrightarrow INV_2[pos := pos+1, data(cb)[pos] := data(cb)[pos+1]]$$

We will refer to the conclusion of this verification condition $INV\_2[pos := pos+1, data(cb)[pos] := data(cb)[pos+1]]$ after the application of substitution as to INV_2'. We instantiate the existential quantifiers in INV_2' for the head and tail lists simply with hd and tl. We instantiate the quantifiers for string intervals a, b, c, d, and e with $a, b, c+1, d-1$, and e, respectively. This reflects the progress of shifting characters in the current block of the string.

With the arithmetic simplification and omitting the trivial conjuncts which follow immediately from the implication assumptions of VC_4 we have to show the following facts.

The first is the existence of string relation:

$$\begin{gathered} String(p, first, next, prev, len, data, hd \odot cb \odot tl, \\ s[0 : a+b-1] \odot s[a+b+1 : a+b+c+1] \odot s[a+b+c+1 : n-1]). \end{gathered}$$

Note that *data* denotes the function after the function update caused by the substitution $[data(cb)[pos] := data(cb)[pos+1]]$.

The second condition we are to show refers to the string interval which corresponds to the current block:

$$\begin{gathered} s[a : a+b-1] \odot s[a+b+1 : a+b+c+1] \odot s[a+b+c+1 : a+b+c+d-1] \\ = data(cb)[0 : pos-1] \odot data(cb)[pos+1] \odot data(cb)[pos+1 : len(cb)-1]. \end{gathered}$$

Actually one sees that the first condition can be reduced to the second one. The reason is straightforward — during the loop execution the head and tail parts of the string stay unchanged. The equalities between implementation and specification sides of these parts are the same in INV_2 and INV_2'. With this fact and the second condition we can show the first condition simply by unfolding the *String* definition. Thus, we continue with the proof of the second condition.

Let us rewrite the equality of the specification and implementation sides of the string part which corresponds to the current blocks from INV_2 as $s_{INV\_2} = data_{INV\_2}$ where:

$$\begin{aligned} s_{INV\_2} &= s[a : a+b-1] \odot s[a+b+1 : a+b+c] \\ &\quad \odot s[a+b+c] \odot s[a+b+c+1 : a+b+c+d-1], \\ data_{INV\_2} &= data(cb)[0 : pos-1] \\ &\quad \odot data(cb)[pos] \odot data(cb)[pos+1 : len(cb)-1]. \end{aligned}$$

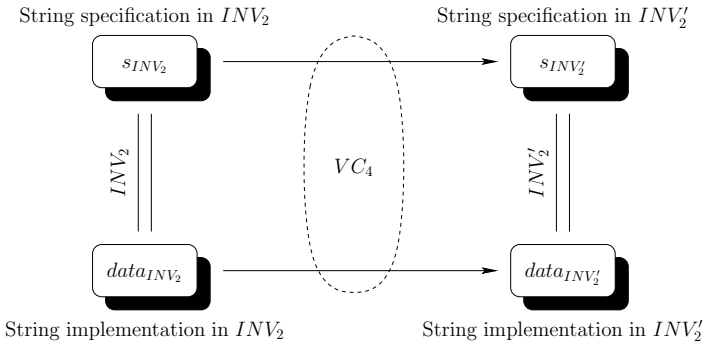

Figure 6.1: Preservation of the string relation during loop iteration

Consider now the diagram in figure 6.1. The equality $s_{INV\_2} = data_{INV\_2}$ transforms through the implication in verification condition VC_4 to the equality of the condition we are showing $s_{INV\_2'} = data_{INV\_2'}$ where:

$$\begin{aligned} s_{INV\_2'} &= s[a:a{+}b{-}1] \odot s[a{+}b{+}1 : a{+}b{+}c] \\ &\quad \odot s[a{+}b{+}c{+}1] \odot s[a{+}b{+}c{+}1 : a{+}b{+}c{+}d{-}1], \\ data_{INV\_2'} &= data(cb)[0 : pos{-}1] \\ &\quad \odot data(cb)[pos{+}1] \odot data(cb)[pos{+}1 : len(cb){-}1]. \end{aligned}$$

Contracting the common parts concatenated on the respective sides of equalities simplifies the implication $s_{INV\_2} = data_{INV\_2} \longrightarrow s_{INV\_2'} = data_{INV\_2'}$ to $s[a{+}b{+}c] = data(cb)[pos] \longrightarrow s[a{+}b{+}c{+}1] = data(cb)[pos{+}1]$ which clearly holds. Thus VC_4 is shown.

6.2.5 From the Second Invariant to the Postcondition

The last verification condition we are to show is:

	VC_5
$INV_2 \wedge pos \geq len(cb){-}1 \longrightarrow POST[len(cb) := len(cb){-}1]$	

Let us consider the case when we do not delete the current block from the string, i.e. it is not empty. The correctness of the other case follows from the correctness of `dList_block_Delete` function and nearly the same kind of reasoning we present here. We start with the quantifier elimination in INV_2 and substitution application to $POST$.

The postcondition $POST$ branches on the value of pos. Since the case $pos \geq n$ is handled in the first verification condition VC_1 we treat here only the latter case, i.e. $pos < n$. We instantiate the existential quantifier in front of the string relation in $POST$ with $hd \odot cb \odot tl$. On the termination of the second loop $pos = len(cb) - 1$ holds. We substitute this equation in the equation for string intervals and therefore conclude $d = 1$. In INV_2 this results in:

$s[a : a+b-1] \odot s[a+b+1 : a+b+c] \odot s[a+b+c] = data(cb)[0 : len(cb)-1]$.

We contract the last character on both sides of this equality and obtain the relation between the implementation and the specification of the string part which corresponds to the current block in $POST$ (after the substitution $[len(cb) := len(cb) - 1]$):

$$s[a : a + b - 1] \odot s[a + b + 1 : a + b + c] = data(cb)[0 : len(cb) - 2].$$

Since the other part of the string stay constant we conclude that the string relation holds in $POST$.

The separation properties in this verification condition are reflected in the following implication to be shown:

$$Pres(cb, \{data\}) \ \wedge \ Pres([\,], \{first, next, prev, len\}) \longrightarrow \\ Pres(p, \{first\}) \ \wedge \ Pres(l_{PRE}, \{next, prev, len, data\})).$$

Again, since we extend the range of the references on which the preservation does not hold, the implication holds.

The remaining facts we have to show, namely $\mathcal{S}(hd \odot cb \odot tl) \subseteq \mathcal{S}(l_{PRE})$ and $res = \texttt{NO\_ERROR}$ follow directly from the assumption INV_2 of the verification condition implication. Hence we conclude VC_5.

6.3 Termination Proof

6.3.1 First Loop

The ranking function for this loop is pos. With every iteration of the loop the position pos is decremented by $len(cb)$. Therefore, the condition for the termination of this loop is:

$TERMINATION\_1$
$pos - len(cb) < pos$

It holds since from the string relation if follows that $len(cb)$ always greater than zero.

6.3.2 Second Loop

The ranking function for this loop is $len(cb) - pos$. The position pos is incremented at every loop's iteration. The termination condition, then, is as follows:

$TERMINATION\_2$
$len(cb) - (pos + 1) < len(cb) - pos$

It obviously holds.

Chapter 7

Summary

In this thesis we presented a sound basis of Hoare Logic rules which underline the formal reasoning about programs written in C0 programming language. These rules are the lite version of the proof system embedded in the Hoare Logic environment by Schirmer [Sch05]. In the spirit of Mehta and Nipkow [MN04] we formalized the notion of references and pointer structures in a way that the Hoare Logic rules stay applicable. We introduced a variety of predicates which allow us to reason about the correctness of complex pointer structures formally. Our definitions of these predicates straightforwardly reflect the recursive definitions in the theories of the computer aided verification environment in Isabelle/HOL [Sch04].

As a next step we described the implementation of a data structure for strings. We presented the formal model of a string which allows to decide whether an implementation of a string meets its specification at any point of program execution. We implemented the desired algorithms on strings and annotated our implementations.

We continued with the formal specification of the functions from the library. We presented these specifications exploiting abstract mathematical notation which, on the one hand, is able to reflect all the basic ideas of the specifications in Isabelle/HOL, and, on the other hand, is sufficiently flexible to omit computer aided details which are hard to read. Each of the loops of these functions was supplemented with a sufficient invariant and a ranking function. The specifications as well as invariants were described in details.

Finally the complete verification process of one of the functions was shown. We pointed out verification conditions obtained by the verification condition generator in the Hoare Logic environment of Isabelle/HOL which applies the Hoare Logic rules automatically. We concluded with the sketches of the proof for these verification conditions. In the thesis we tried to show only the key ideas of the verification of algorithms on pointer structures. Therefore our paper-and-pencil proofs should be understood as skeletons for the full proofs in Isabelle/HOL which involve much more formal details.

Bibliography

[Apt81] K.R. Apt. *Ten Years of Hoare's Logic: A Survey – Part I.* ACM Transactions on Programming languages and Systems, vol. 3, no. 4, Oct 1981, pp. 431-483.

[Bor00] R. Bornat. *Proving Pointer Programs in Hoare Logic.* Mathematics of Program Construction, vol. 1837 of LNCS, Springer-Verlag, 2000, pp. 102-106.

[Bur72] R. Burstall. *Some techniques for proving correctness of programs which alter data structures.* Machine Intelligence 7, Edinburgh University Press, 1972, pp. 23-50.

[C99] *Programming languages — C.* International Standard ISO/IEC ISO/IEC 9899:1999 (E).

[Cla79] E.M. Clarke jr. *Programming Language Constructs for Which It Is Impossible to Obtain Good Hoare Axiom Systems.* Journal of the ACM, vol. 26, no. 1, Jan 1979, pp. 129-147.

[CLR03] T.H. Cormen, C.E. Leiserson, R.L. Rivest *Introduction to Algorithms. Second Edition.* The MIT Press, Cambridge-London, 4th printing, 2003.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Hoa69] C.A.R. Hoare. *An Axiomatic Basis for Computer Programming.* Communications of the ACM, vol. 12, no. 10, Oct 1969, pp. 576-580, 583.

[Hoa71] C.A.R. Hoare. *Procedures and parameters: An axiomatic approach.* Lecture Notes in Mathematics, vol. 188, Springer-Verlag, New York, 1971, pp. 102-116.

[LPP05] D. Leinenbach, W.J. Paul, and E. Petrova *Towards the Formal Verification of a C0 Compiler.* In Proceedings, 3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), Koblenz, Germany, Sep 5-9, 2005.

[MN04] F. Mehta, T. Nipkow. *Proving Pointer Programs in Higher-Order Logic.* Automated Deduction, vol. 2741, Springer-Verlag, 2003, pp. 121-135.

[Ngu05] V.G. Nguiekom. *Verifikation von doppelt verketteten Listen auf Pointerebene* Diplomarbeit, Universitaet des Saarlandes, Mai 2005.

[NPW04] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* Springer-Verlag, 2004.

[Pre05] H. Prediger. *Formal Verification of a C-Library for Strings.* Diploma Thesis, Saarland University, Jul 2005.

[Pop03] N. Popov. *Verification Using Weakest Precondition Strategy.* Contributed talk at Computer Aided Verification of Information Systems (CAVIS-04), Timisoara, Romania, Feb 2003.

[Sch04] N. Schirmer. *Hoare Logic 1.7.* Isabelle theories, TU Muenchen, Aug 2004.

[Sch05] N. Schirmer. *A Verification Environment for Sequential Imperative Programs in Isabelle/HOL.* Logic for Programming, Artificial Intelligence, and Reasoning, Montevideo, Uruguay, Mar 14-18, 2005.

[Ste03] W. Stephan *Verification of Pointer Programs in Dynamic Logic.* Internal Technical Report 7, Verisoft Project, www.verisoft.de, Dec 2003.