Formulas as Programs

Eva Suci, Master Thesis, Spring 2003

Supervisor: Marc Bezem

Contents

Chapter 1

Introduction

Declarative programming is a style of programming in which a program has a dual reading both as a sequence of instructions to a machine and as a formula in a logic with a simple semantics. The execution of the program should respect the logical semantics. Examples (with the corresponding logics) are: functional programming (equational logic), logic programming (predicate logic), constraint logic programming (predicate logic over specific domains) and database querying (set theory). The dual reading of a program as a formula is - deliberately - more abstract, abstracting for example from a computational notion like state. This makes declarative programs easier to understand, modify and verify, which - ideally - should outweigh the lesser expressivity and efficiency.

Apt and Bezem [1] introduced a computational interpretation of first-order logic based on a constructive interpretation of satisfiability w.r.t. a fixed but arbitrary interpretation, in which approach the formulas themselves are programs.

The goal of this project has been to explore declarative programming by extending this interpretation to such Alma-0 programs which contain recursive and/or non-recursive procedures but still do not include destructive assignments, as well as to give a formal proof for the implication:

$$\mathit{Implementation} \rightarrow \mathit{Specification}$$

for some non-trivial examples.

Alma-0 is an implemented programming language that supports declarative programming, and which combines the advantages of imperative and logic programming paradigms.

Using Alma-0, we have elaborated declarative implementations for well-known non-trivial computational problems, such as: finding a maximum element of an array and sorting, accompanied by formal proofs for the implication above.

Subsections of this chapter give a general overview of imperative, logic and functional programming, pointing out main traits and differences of these

programming paradigms. [11]

1.1 Imperative Programming

The basic architecture of computers has had a crucial effect on programming and language design. Most of the popular languages of the past decades (i.e. Fortran, Pascal, C, etc.), have been designed around the dominant computer architecture, called the von Neumann architecture (after one of its originators). In such a computer both data and programs are stored in the same memory, whereas the central processing unit (CPU), which executes instructions, is separated from the memory, therefore data flow and instruction transfer between the CPU and memory must be assured by some other means. Nearly all digital computers since '40s have been based on this architecture.

Imperative programming languages are, to varying degrees, abstractions of the underlying von Neumann computer. The abstractions in a language for the memory cells of the machine are variables. In some cases, the characteristics of the abstractions are very close to the characteristics of the cells. For example, an integer variable is usually represented exactly as an individual hardware memory word. In other cases, the abstractions are at a large distance from the cells, as with a three-dimensional array, which requires a software mapping function to support the abstraction.

The central features of imperative programming languages are variables together with the assignment statements and control statements. The purpose of an assignment statement is to change the value of a variable, so an integral part of imperative programming is the concept of variables whose values change during program execution. (Non-imperative languages may include variables of a different sort, such as the parameters of functions in functional languages.)

An assignment statement can simply cause a value to be copied from one memory cell to another. But in many cases assignment statements include expressions with operators. In this case, operands in expressions are piped from memory to the CPU, and the result of evaluating the expression is piped back to the memory cell represented by the left side of the assignment.

Control statements are linguistic mechanisms that make the computations in imperative programs flexible and powerful. There are basically two main mechanisms: some means of selecting among alternative control flow paths (of statement execution) and some means of causing the repeated execution of certain collections of statements.

The most efficient way to implement repetition on von Neumann computers is the iterative form because in iteration instructions are stored in adjacent memory cells. This efficiency discourages the use of recursion for repetition, although recursion is often more natural.

The most common control statements present in imperative languages are:

if_ then_ else as a form of selection statement, the *for* loop as a finite iterative form of repetition, and the *while* statement which is a logically controlled loop, possibly infinite. There are of course many others which may differ in nuances from language to language.

Programming in imperative languages is primarily procedural, which means that the programmer instructs the computer exactly how a certain computation should be accomplished. That is, the computer is treated as a simple device that obeys orders. Everything that is computed must have every detail of the computation spelled out.

For example, if we have a glass of water and a glass of milk, we know that in order to be able to swap the content of the glasses we need a third empty glass. In the case of a procedure that swaps two integers that's exactly what happens:

```
PROCEDURE Swap(VAR x,y:INTEGER);
VAR z:INTEGER;
BEGIN
  z := x;
  y := x;
  x := z;
END;
```

We communicated every detail needed for swapping: the request for a new variable z, and the set of instructions that do the work.

1.2 Logic Programming

Logic programming is the use of a formal logic notation to communicate computational processes to a computer. A subset of predicate calculus, called Horn clause logic, is the notation used in current logic programming languages. A logic programming language is a rule-based language, where rules are specified in no particular order, and the language implementation system must choose an execution order that produces the desired result.

Non-imperative languages, and in particular logic programming languages are nonprocedural. Programs in such languages, unlike programs in imperative languages, do not state exactly how a result is to be computed but rather describe what result. The difference is that we assume the computer system can somehow determine how the result is to be computed. What is needed to provide this capability for logic programming languages is a concise means of supplying the computer with both the relevant information and an inferencing process for computing desirable results. Predicate calculus supplies the basic form of communication to the computer, and the proof method, named resolution [10], supplies the inference technique. For Horn clause logic this specialised inference method is called SLD-resolution, which forms the basis for the Prolog programming language.

In fact, resolution is the primary activity of a Prolog interpreter. This process, which uses backtracking extensively, involves mainly pattern matching among propositions. When variables are involved, they can be instantiated to values to provide matches. This instantiation process is called *unification*.

Languages used for logic programming are called *declarative languages* because programs written in them consist of declarations rather then assignments and control flow statements. These declarations are actually statements, or propositions, in symbolic logic. In this context, logic programming abstracts from a computational notion like machine state, the execution of the program being characterised by the inference rules.

One of the essential characteristics of logic programming languages is their semantics, which is called *declarative semantics*. The basic concept of this semantics is that there is a simple way to determine the meaning of each statement, which doesn't depend on how the statement might be used to solve a problem. Therefore declarative semantics are considerably easier to understand than the semantics of imperative languages. For example, the meaning of a given proposition in a logic programming language can be concisely determined from the statement itself, whereas in an imperative language, the semantics of a simple assignment statement requires examination of local declarations, knowledge of the scoping rules of the language, possibly even examinations of programs in other files just to determine the types of variables in the assignment, and so on. Thus, declarative semantics, with no need to consider textual context or execution sequences, is often stated as one of the advantages that declarative languages have over imperative languages.

For example in Prolog the above mentioned swapping problem could be specified as a relation of arity 4:

```
SWAP(X,Y,Y,X)
```

There are a number of problems with the current state of logic programming. For reasons of efficiency, and even to avoid infinite loops, programmers must sometimes state control flow information in their programs. Also, there are the problems of the so called closed-world assumption (any query about which there is not enough information to prove is assumed to be false) and negation.

Logic programming has been used in a number of different areas, primarily in relational database systems, expert systems, and natural language processing.

1.3 Functional Programming

The functional programming paradigm, which is based on mathematical functions, is the design basis for one of the most important non-imperative styles

of languages. This style of programming is supported by functional, or applicative, programming languages.

The objective of the design of functional programming language is to mimic mathematical functions to the greatest possible. This results in an approach to problem solving that is fundamentally different from methods used in imperative languages.

As we have seen, in imperative programming, an expression is evaluated and the result is stored in a memory location, which is represented by a variable in the program. A purely functional programming language does not use variables or assignment statements. This frees the programmer from concerns about the memory cells of the computer on which the program is executed. Without variables, iterative constructs are not possible, for they are controlled by variables. Therefore, iteration must be done by recursion rather then by repetition.

Programs are function definitions and function application specifications, and executions consist of evaluating the function applications.

In functional programming, for example, the before mentioned swapping could be specified as follows:

```
FUN swap(x,y :  INT) = (y,x) END
```

Without variables, the execution of a purely functional program has no state in the sense of operational and denotational semantics. The execution of a function always produces the same result when given the same parameters, this property being called *referential transparency*. It makes the semantics of purely functional languages far simpler than the semantics of the imperative languages and functional languages that include imperative features.

A functional language provides a set of primitive functions, a set of functional forms to construct complex functions from these primitive functions, a function application operation, and some structures for representing data. These structures are used to represent parameters and values computed by functions. A well-defined functional language requires only a small number of primitive functions.

Functional languages can have a very simple syntactic structure. The list structure of LISP is an example. LISP began as a purely functional language but soon required some important imperative features that increased its execution efficiency.

ML is a strongly typed functional language with more conventional syntax than LISP, one that is more similar to Pascal. It includes a type inferencing system and exception handling.

Although there may be advantages to purely functional languages over their imperative relatives, their lower efficiency of execution on von Neumann computers has prevented them from being considered by many as substitutes.

Chapter 2

Operational Semantics

As a starting point we used the computation mechanism introduced by Apt and Bezem [1], and analysed the soundness and completeness results obtained there. As the defined language was too limited as a formalism for programming, they discuss some possible extensions convenient for programming purposes, such as: non-recursive procedures, sorts (i.e. types), arrays and bounded quantification.

In this chapter we are presenting this computation mechanism, spelled out and compiled it with the extensions there introduced, extending it further on for recursive procedures, and according to all these, soundness and completeness results are recovered.

2.1 Computation Mechanism

Consider an arbitrary many-sorted first-order language with equality and an interpretation for it. Interpretations for many-sorted first-order languages are obtained by assigning to each sort a non-empty domain and by assigning to each function symbol and each predicate symbol, respectively, an appropriate function and relation on these sorts. We assume in particular a fixed signature with a corresponding interpretation of its elements in the domains. Arities in the signature specify the sorts of the arguments of the function and predicate symbols, as well as the sorts of the function values. Terms and atoms are well-formed if the sorts of the arguments comply with the signature. In quantifying a variable, its sort, if not clear from the context, should be made explicit.

Sorts can be used to model various basic data types occurring in programming practice: integers, booleans, characters, but also compound data types such as arrays.

Arrays are modelled as vectors or matrices, using projection functions that are given a *standard interpretation.* Given a sort for the indices (typically, a segment of integers or a product of such segments) and a sort for the elements of the array, we add a sort for arrays of the corresponding type to the signature. We also add to the language *array variables*, or *arrays* for short, to be

interpreted as arrays in the standard interpretation.

We use the letters a, b, c to denote arrays and to distinguish arrays from objects of other sorts. We write $a[t_1, t_2, ..., t_n]$ to denote the projection of the array a on the index $t_1, t_2, ..., t_n$, akin to the use of subscripted variables in programming languages. The standard interpretation of each projection function maps a given array and a given index to the correct element. Thus subscripted variables are simply terms, but they are handled in such a way, that, e.g., $a[t_1, t_2, ..., t_n]$ is viewed as a variable and not as a compound term.

Definition 1 (valuation, α-closed, α-assignment). A *valuation* is a finite mapping from variables to domain elements. Valuations will be denoted as single-valued sets of pairs x/d, where x is a variable and d a domain element. We use $\alpha, \alpha', \beta, \beta', ...$ for arbitrary valuations and call α' an *extension* of α when $\alpha \subseteq \alpha'$, that is, every assignment to a variable by α also occurs in α'. Further, ε denotes the empty valuation. A variable x is α-*closed*, if for some d the pair x/d is an element of α.

Let α be a valuation. A term t is α-*closed*, if all variables of t get a value in α. In that case t^α denotes the *evaluation* of t under α in the domain. More generally, for any expression E the result of the replacement of each α-*closed* term t by t^α is denoted by E^α.

In general, we say that a term $f(t_1, t_2, ..., t_n)$, with f function symbol, is α-*closed* if each term t_i is α-*closed*.

As an extension to this valuation we define the *array valuation* to be a finite mapping from array elements to domain elements. They will be denoted as pairs of the form $a[d_1, d_2, ..., d_n]/d$, where a is an n-ary array symbol, and $d_1, d_2, ..., d_n, d$ are domain elements. This means that, if the terms $t_1, t_2, ..., t_n$ evaluates to $d_1, d_2, ..., d_n$ respectively, then the term $a[t_1, t_2, ..., t_n]$, handled rather as a variable, evaluates to d.

A term $a[t_1, t_2, ..., t_n]$ is α-*closed,* if each term t_i is α-*closed* and evaluates to a domain element d_i such that for some d the pair $a[d_1, d_2, ..., d_n]/d$ is an element of α.

Hereafter, whenever we will refer to a *valuation* α, we will consider its extension to arrays as well.

An equation $s = t$ is an α-*assignment* if either

- one side of it, say s, is a variable that is not α-*closed* and the other side, t, is an α-*closed* term, or
- one side of it, say s, is of the form $a[t_1, t_2, ..., t_n]$, where each t_i is α-*closed* but $a[t_1, t_2, ..., t_n]$ is not α-*closed*, and the other, t, is an α-*closed* term.

In our setting, the only way to assign values to variables or arrays at a selected position will be by evaluating an α-assignment as above. Given an α-assignment $x = t$, we evaluate it by assigning to x the value t^α, or given an α-assignment $a[t_1, t_2, ..., t_n] = t$, we evaluate it by assigning $a[t_1^\alpha, t_2^\alpha, ..., t_n^\alpha]$ the

value t^α.

Definition 2 (formulas). In order to accommodate the definition of the operational semantics, the set of formulas has an inductive definition which may look a bit peculiar. First, unbounded universal quantifiers are absent. Second, every formula is taken to be a conjunction, with every conjunct (if any) either an atomic formula (in short: an *atom*), or a disjunction, conjunction or implication of formulas, a negation of a formula, an existentially quantified formula, an existentially or universally bounded formula. The latter three unary constructors are assumed to bind stronger then the previous binary ones.

An *atom* is either:

- an equation of the form $s = t$, with s and t terms, or
- a term $p(t_1, t_2, ..., t_n)$, with p, an n-ary relation symbol, and $t_1, t_2, ..., t_n$ terms.

For maximal clarity we give here an inductive definition of the set of formulas. In the operational semantics all conjunctions are taken to be right associative.

1. The empty conjunction $\Box$ is a formula.
2. If ψ is a formula and A is an atom, then $A \wedge \psi$ is a formula.
3. If ψ, ϕ_1, ϕ_2 are formulas, then $(\phi_1 \vee \phi_2) \wedge \psi$ is a formula.
4. If ψ, ϕ_1, ϕ_2 are formulas, then $(\phi_1 \wedge \phi_2) \wedge \psi$ is a formula.
5. If ψ, ϕ_1, ϕ_2 are formulas, then $(\phi_1 \rightarrow \phi_2) \wedge \psi$ is a formula.
6. If ϕ, ψ are formulas, then $\neg\phi \wedge \psi$ is a formula.
7. If ϕ, ψ are formulas, then $\exists x\, \phi \wedge \psi$ is a formula.
8. If $\phi(x)$ is a formula, with x of integer type, and s and t are terms of integer type, and ψ is a formula, then $\exists x \in [s..t]\, \phi(x) \wedge \psi$ is a formula.
9. If $\phi(x)$ is a formula, with x of integer type, and s and t are terms of integer type, and ψ is a formula, then $\forall x \in [s..t]\, \phi(x) \wedge \psi$ is a formula.

If ϕ is a formula and α is a valuation, then we say that ϕ is α-closed, if for every free variable x of ϕ and some domain element d, the pair x/d is an element of α.

Definition 3 ($\wp$-axiom). If p is an n-ary relation symbol, and ψ is a formula with free variables $x_1, x_2, ..., x_n$, then the equivalence

$$p(x_1, x_2, ..., x_n) \leftrightarrow \psi$$

is called a $\wp$-*axiom*, where the equivalence $\leftrightarrow$connects the weakest with respect to any other connectives. A $\wp$-axiom is always considered to be universally closed w.r.t. the free variables $x_1, x_2, ..., x_n$.

As we did not impose any restriction on ψ, apart from claiming $x_1, x_2, ..., x_n$ to be free variables in ψ, such an axiom may not properly define p or even be flat-out inconsistent, when, for example, ψ itself contains an atom of the form $p(t_1, t_2, ..., t_n)$.

The reader may consult several examples:

- $p(x, y, z) \leftrightarrow (x \leq y \wedge y = z) \vee (x > y \wedge x = z)$ is well-defined
- $odd(x) \leftrightarrow x = 1 \vee (x > 0 \wedge even(x - 1))$
  and $even(x) \leftrightarrow x = 0 \vee (x > 0 \wedge odd(x - 1))$
  $odd(x)$ and $even(x)$ are mutually recursive, well-defined on $\mathbb{N}$
- $p(x) \leftrightarrow p(x)$ is pathological: consistent, but p is ill-defined
- $p(x) \leftrightarrow (p(x) \rightarrow 0 = 2)$ is pathological: inconsistent
- $p(x) \leftrightarrow \neg p(x)$ is pathological: inconsistent.

A $\wp$-axiom $p(x_1, x_2, ..., x_n) \leftrightarrow \psi$ is called *consistent*, when p, if present in ψ at all, doesn't occur in the scope of a negation or the premise part of an implication, that is, p appears only in strictly positive positions in ψ.

Hereafter, we tacitly assume that all $\wp$-axioms are consistent. In our interpretation, under the assumption of consistency, all such definitions become true.

In general, the logical reading of an implementation $\mathbf{I}$ consists of a conjunction of finitely many $\wp$-axioms (all with different relation symbols on the left-hand side) and of a formula ϕ representing the main body of the implementation. This can be made rigorous by the following syntax:

$$\mathbf{I} ::= \wp\text{-axiom} \wedge \mathbf{I} \mid \phi \tag{2.1}$$

where the relation symbol on the left-hand side in the newly added $\wp$-axiom is always fresh with respect to the the left-hand sides of the $\wp$-axioms already added.

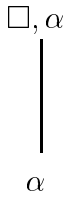
For a given implementation defined as in (2.1), we denote by $\wp_I$ the conjunction of $\wp$-axioms, and by ϕ_I the formula, or when no ambiguity arises, $\wp$ and ϕ, respectively.

Definition 4 (operational semantics). The operational semantics of a formula will be defined in terms of a possibly infinite tree $[\![\phi]\!]_\alpha$ depending on the formula ϕ, the (initial) valuation α, and the associated $\wp$ in the given implementation. The root of $[\![\phi]\!]_\alpha$ is labelled with the pair ϕ, α. All internal nodes of the tree $[\![\phi]\!]_\alpha$ are labelled with pairs consisting of a formula and a valuation. The leaves of the tree are labelled with either

- *error* (representing the occurrence of an error in this branch of the computation), or
- *fail* (representing logical failure of the computation), or
- a valuation (representing logical success of the computation and yielding values for the free variables of the formula that make the formula true relative to $\wp$).

Definition 5 (computation tree). The computation tree $[\![\phi]\!]_\alpha$ is defined by using the structure of *formulas* given by Definition 2.

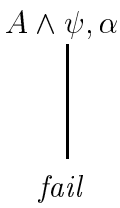
1. For the empty conjunction we define $[\![\Box]\!]_\alpha$ to be the tree with the root that has a success leaf α as its son:



2. If ψ is a formula and A is an atom, then we distinguish ten cases depending on the form of A. In all the ten cases $[\![A \wedge \psi]\!]_\alpha$ is a tree with a root of degree one.

   - Atom A has the form $s = t$, is α-closed and true. Then the root of $[\![A \wedge \psi]\!]_\alpha$has $[\![\psi]\!]_\alpha$ as its subtree:

   - Atom A has the the form $s = t$, is α-closed and false. Then the root of $[\![A \wedge \psi]\!]_\alpha$has the failure leave *fail* as its son:



   - Atom A has the the form $s = t$, is not α-closed, but is not an

α-assignment. Then the root of $[\![A \wedge \psi]\!]_\alpha$has the *error* leave as its son:

$$A \wedge \psi, \alpha$$
$$|$$
$$error$$

- Atom A is an α-assignment $s = t$, such that either s or t is a variable which is not α-closed, say $s \equiv x$, with x not α-closed and t α-closed. Then the root of $[\![A \wedge \psi]\!]_\alpha$has the $[\![\psi]\!]_{\alpha'}$ as its subtree, where α' extends α with the pair x/t^α:

$$A \wedge \psi, \alpha$$
$$|$$
$$[\![\psi]\!]_{\alpha'}$$

  The symmetrical case is analogous.

- Atom A is an α-assignment $s = t$, such that s or t, say s, is a term of the form $a[t_1, t_2, ..., t_n]$, with each t_i α-*closed* but $a[t_1, t_2, ..., t_n]$ not α-*closed*, and t α-closed. Then the root of $[\![A \wedge \psi]\!]_\alpha$has $[\![\psi]\!]_{\alpha'}$ as its subtree, where α' extends α with the pair $a[t_1^\alpha, t_2^\alpha, ..., t_n^\alpha]/t^\alpha$:

$$A \wedge \psi, \alpha$$
$$|$$
$$[\![\psi]\!]_{\alpha'}$$

  The symmetrical case is analogous.

- Atom A has the form $p(t_1, t_2, ..., t_n)$, where p is a predefined n-ary relation symbol, $t_1, t_2, ..., t_n$ are α-closed, and $p(t_1^\alpha, t_2^\alpha, ..., t_n^\alpha)$ is true, then the root of $[\![A \wedge \psi]\!]_\alpha$ has $[\![\psi]\!]_\alpha$ as its subtree.

$$p(t_1, t_2, .., t_n) \wedge \psi, \alpha$$
$$|$$
$$[\![\psi]\!]_\alpha$$

(Typically some binary predefined relation symbols: $<$, $\leq$, $>$, $\geq$)

- Atom A has the form $p(t_1, t_2, ..., t_n)$, where p is a predefined n-ary relation symbol, $t_1, t_2, ..., t_n$ are α-closed, and $p(t_1^\alpha, t_2^\alpha, ..., t_n^\alpha)$ is false, then the root of $[\![A \wedge \psi]\!]_\alpha$ has the failure leaf *fail* as its son.

$$p(t_1, t_2, .., t_n) \wedge \psi, \alpha$$
$$|$$
$$\mathit{fail}$$

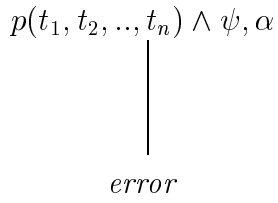
- Atom A has the form $p(t_1, t_2, ..., t_n)$, where p is a predefined n-ary relation symbol, and at least one term from $t_1, t_2, ..., t_n$ is not α-closed, then the root of $[\![A \wedge \psi]\!]_\alpha$ has the error leaf *error* as its son.

$$p(t_1, t_2, .., t_n) \wedge \psi, \alpha$$
$$|$$
$$\mathit{error}$$

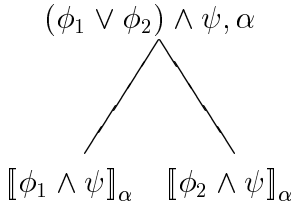
- Atom A has the form $p(t_1, t_2, ..., t_n)$, where p is an n-ary relation symbol, but not predefined. If $\wp$ contains a $\wp$-axiom for p, such that $p(x_1, x_2, ..., x_n) \leftrightarrow \phi$, then the root of $[\![A \wedge \psi]\!]_\alpha$ has $[\![\phi\{x_1/t_1, ..., x_n/t_n\} \wedge \psi]\!]_\alpha$ as its subtree, where $\phi\{x_1/t_1, x_2/t_2, ..., x_n/t_n\}$ stands for the result of substituting in ϕ the free occurrences of variables $x_1, x_2, ..., x_n$ by $t_1, t_2, ..., t_n$ respectively:

$$p(t_1, t_2, .., t_n) \wedge \psi, \alpha$$
$$|$$
$$[\![\phi\{x_1/t_1, ..., x_n/t_n\} \wedge \psi]\!]_\alpha$$

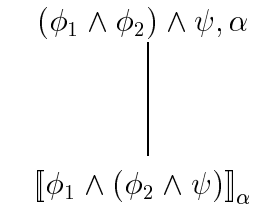
- Atom A has the form $p(t_1, t_2, ..., t_n)$, where p is an n-ary relation symbol, but not predefined. If $\wp$ doesn't contain a $\wp$-axiom for p, then the root of $[\![A \wedge \psi]\!]_\alpha$ has the *error* leave as its son:



3. If ψ, ϕ_1, ϕ_2 are formulas, then we put $[\![(\phi_1 \vee \phi_2) \wedge \psi]\!]_\alpha$to be the tree with a root of degree two and with left and right subtrees $[\![\phi_1 \wedge \psi]\!]_\alpha$ and $[\![\phi_2 \wedge \psi]\!]_\alpha$, respectively:



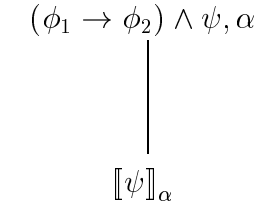
4. If ψ, ϕ_1, ϕ_2 are formulas, then we put $[\![(\phi_1 \wedge \phi_2) \wedge \psi]\!]_\alpha$ to be the tree with a root of degree one with $[\![\phi_1 \wedge (\phi_2 \wedge \psi)]\!]_\alpha$ as its subtree:



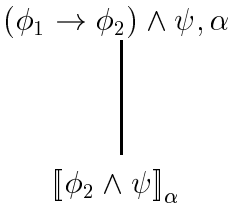
   This substantiates the association of conjunctions to the right as mentioned in Definition 2.

5. If ψ, ϕ_1, ϕ_2 are formulas, then we put $[\![(\phi_1 \rightarrow \phi_2) \wedge \psi]\!]_\alpha$to be the tree with a root of degree one. We distinguish three cases.

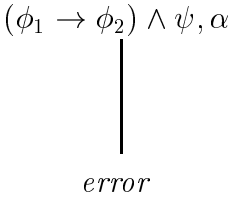
   - Formula ϕ_1 is α-closed and $[\![\phi_1]\!]_\alpha$ is finite and contains only failure leaves. Then the root of $[\![(\phi_1 \rightarrow \phi_2) \wedge \psi]\!]_\alpha$ has $[\![\psi]\!]_\alpha$ as its subtree:



   - Formula ϕ_1 is α-closed and $[\![\phi_1]\!]_\alpha$ contains at least one success leaf. Then the root of $[\![(\phi_1 \rightarrow \phi_2) \wedge \psi]\!]_\alpha$ has $[\![\phi_2 \wedge \psi]\!]_\alpha$ as its subtree:



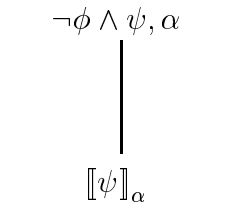
- In all the other cases the root of $[\![(\phi_1 \to \phi_2) \wedge \psi]\!]_\alpha$ has the error leaf *error* as its son:



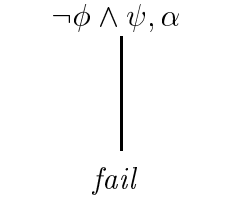
The above definition relies on the logical equivalence of $\phi_1 \to \phi_2$ and $\neg\phi_1 \vee \phi_2$, but avoids unnecessary branching in the computation tree that would be introduced by the disjunction.

6. If ϕ, ψ are formulas, then to define $[\![\neg\phi \wedge \psi]\!]_\alpha$ we distinguish three cases w.r.t. ϕ. In all of them $[\![\neg\phi \wedge \psi]\!]_\alpha$ is a tree with a root of degree one.

   - Formula ϕ is α-closed and $[\![\phi]\!]_\alpha$ is finite and contains only failure leaves. Then the root of $[\![\neg\phi \wedge \psi]\!]_\alpha$ has $[\![\psi]\!]_\alpha$ as its subtree:



   - Formula ϕ is α-closed and $[\![\phi]\!]_\alpha$ contains at least one success leaf. Then the root of $[\![\neg\phi \wedge \psi]\!]_\alpha$ has the failure leaf *fail* as its son:



   - In all other cases the root of $[\![\neg\phi \wedge \psi]\!]_\alpha$ has the error leaf *error* as its son:

$$\neg\phi \wedge \psi, \alpha$$
$$|$$
$$error$$

There are basically two classes of formulas ϕ in this contingency: those that are not α-closed, and those for which $[\![\phi]\!]_\alpha$ contains no success leaf, but either it contains an error leaf or an infinite path or both.

7. The case of $\exists x\ \phi \wedge \psi$ requires the usual care with bound variables to avoid name clashes. Let α be a valuation. First, we require that the variable x does not occur in the domain of α. Second, we require that the variable x does not occur in ψ. Both requirements are summarized by phrasing that x is *fresh* with respect to α and ψ. They can be met by appropriately renaming the bound variable x.

   With x fresh as above we define $[\![\exists x\ \phi \wedge \psi]\!]_\alpha$ to be the tree with a root of degree one and $[\![\phi \wedge \psi]\!]_\alpha$ as its subtree:

$$(\exists x\ \phi) \wedge \psi, \alpha$$
$$|$$
$$[\![\phi \wedge \psi]\!]_\alpha$$

   Thus the operational semantics of $\exists x\ \phi \wedge \psi$ is, apart from the root of degree one, identical to that of $\phi \wedge \psi$. This should not come as a surprise, as $\exists x\ \phi \wedge \psi$ is logically equivalent to $\exists x\ (\phi \wedge \psi)$ when x does not occur in ψ. Observe that success leaves of $[\![\phi \wedge \psi]\!]_\alpha$, and hence of $[\![\exists x\ \phi \wedge \psi]\!]_\alpha$, may or may not contain an assignment for x. For example, $\exists x\ x = 3 \wedge \psi$ yields an assignment for x, but $\exists x\ 3 = 3 \wedge \psi$ does not. In any case the assignment for x is not relevant for the formula as a whole, as the bound variable x is assumed to be fresh. In an alternative approach, the possible assignment for x could be deleted.

8. In the case of $\exists x \in [s..t]\ \phi(x) \wedge \psi$, we require again that the variable x does not occur in the domain of α. We distinguish three cases. In each case the root of $[\![\exists x \in [s..t]\ \phi(x) \wedge \psi]\!]_\alpha$ has a root of degree one and depends on s and t in the following way:

   - If s or t is not α-closed, then the root of $[\![\exists x \in [s..t]\ \phi(x) \wedge \psi]\!]_\alpha$has the error leaf *error* as its son.

$$\exists x \in [s..t]\ \phi(x) \wedge \psi, \alpha$$
$$|$$
$$error$$

- If s and t are α-closed and $s^\alpha > t^\alpha$, then the root of $[\![\exists x \in [s..t]\ \phi(x) \wedge \psi]\!]_\alpha$ has the failure leaf *fail* as its son:

$$\exists x \in [s..t]\ \phi(x) \wedge \psi, \alpha$$
$$|$$
$$fail$$

- If s and t are α-closed and $s^\alpha \leq t^\alpha$, then the root of $[\![\exists x \in [s..t]\ \phi(x) \wedge \psi]\!]_\alpha$ has $[\![(\phi(x) \vee \exists y \in [s{+}1..t]\ \phi(y)) \wedge \psi]\!]_{\alpha \cup \{x/s^\alpha\}}$ as its son, where y is a fresh variable with respect to α, in order to avoid name clashes:

$$\exists x \in [s..t]\ \phi(x) \wedge \psi, \alpha$$
$$|$$
$$[\![(\phi(x) \vee \exists y \in [s{+}1..t]\ \phi(y)) \wedge \psi]\!]_{\alpha \cup \{x/s^\alpha\}}$$

9. In the case of $\forall x \in [s..t]\ \phi(x) \wedge \psi$, we require that the variable x does not occur in the domain of α. We distinguish three cases. In each case the root of $[\![\forall x \in [s..t]\ \phi(x) \wedge \psi]\!]_\alpha$ has a root of degree one and depends on s and t in the following way:

   - If s or t is not α-closed, then the root of $[\![\forall x \in [s..t]\ \phi(x) \wedge \psi]\!]_\alpha$has the error leaf *error* as its son.

$$\forall x \in [s..t]\ \phi(x) \wedge \psi, \alpha$$
$$|$$
$$error$$

   - If s and t are α-closed and $s^\alpha > t^\alpha$, then the root of $[\![\forall x \in [s..t]\ \phi(x) \wedge \psi]\!]_\alpha$ has a success leaf α as its son:

$$\forall x \in [s..t]\ \phi(x) \wedge \psi, \alpha$$
$$\Big|$$
$$\alpha$$

- If s and t are α-closed and $s^\alpha \leq t^\alpha$, then the root of $[\![\forall x \in [s..t]\ \phi(x) \wedge \psi]\!]_\alpha$ has $[\![(\phi(x) \wedge \forall y \in [s{+}1..t]\ \phi(y)) \wedge \psi]\!]_{\alpha \cup \{x/s^\alpha\}}$ as its son, where y is a fresh variable with respect to α, in order to avoid name clashes:

$$\forall x \in [s..t]\ \phi(x) \wedge \psi, \alpha$$
$$\Big|$$
$$[\![(\phi(x) \wedge \forall y \in [s{+}1..t]\ \phi(y)) \wedge \psi]\!]_{\alpha \cup \{x/s^\alpha\}}$$

This computation mechanism with the extensions now included, especially the 9th case of the second clause brings naturally the question of what parameter-passing mechanism should be used when a $\wp$-axiom is replaced by its body, or using terminology from programming, when a procedure is called with its actual parameters.

Consider the following example which illustrates well that depending on the parameter-passing technique the computation mechanism is implemented with, we may obtain different computation trees, that is, the program's behaviours differ and so do then the computed results.

$$(p(x,y,z,w) \leftrightarrow (x < z \wedge w = x) \vee (x \geq z \wedge w = y)) \wedge p(10 * 10, 1\, Div\, 0, 0, w)$$

where $*$ and Div are predefined function symbols for multiplication and division, respectively.

Say, that the computation mechanism upon encountering the atom $p(10 * 10, 1\, Div\, 0, 0, w)$ first evaluates every α-closed argument, and then, when applying the $\wp$-axiom, these values would be assigned to the corresponding free variables (where values are available, of course, otherwise the occurrences of free variables are replaced by actual terms). In this case, because of division by zero, instantly a run-time error will arise, without having even called the body of p.

On the other hand, if the computation mechanism upon encountering the atom $p(10 * 10, 1\, Div\, 0, 0, w)$ just substitutes the occurrences of free variables x, y, z, w with the expressions $10 * 10, 1\, Div\, 0, 0, w$, and simply proceeds, then the computation tree will yield a success leaf $\{w/100\}$ which validates the formula.

In Chapter 3 we reflect to some extent on the parameter-passing mechanism of Alma-0 when presenting briefly some general aspects of the language.

Definition 6 (status of a computation tree). A computation tree is:

- *determined*, if it is either:
  - successful, when it contains at least one success leaf, or
  - failed, when it is finite and contains only failure leaves,
- *undetermined* otherwise, that is, if it contains no success leaf, but either it contains an error leaf or an infinite path (or both).

2.2 Examples

To make all the definitions and concepts of the previous section more concrete, the reader may consult several cases.

Example 1. Consider the following implementation:

$$(p(x) \leftrightarrow x = 2 \vee x = 3) \wedge p(y) \wedge p(z) \wedge y \neq z$$

The computation tree for the formula $\phi = p(y) \wedge p(z) \wedge y \neq z$, with the empty valuation ε, and the associated $\wp$-axiom $p(x) \leftrightarrow x = 2 \vee x = 3$ can be found on the next page.

The two valuations obtained as success leaves yield values for the free variables y and z in ϕ that make ϕ true relative to $\wp$, as the evaluation of ϕ under (any) α depends on $\wp$. This can be made explicit by the following formalism: if α is one of the two success leaves, then:

$$\models \wp \rightarrow \phi^{\alpha}$$

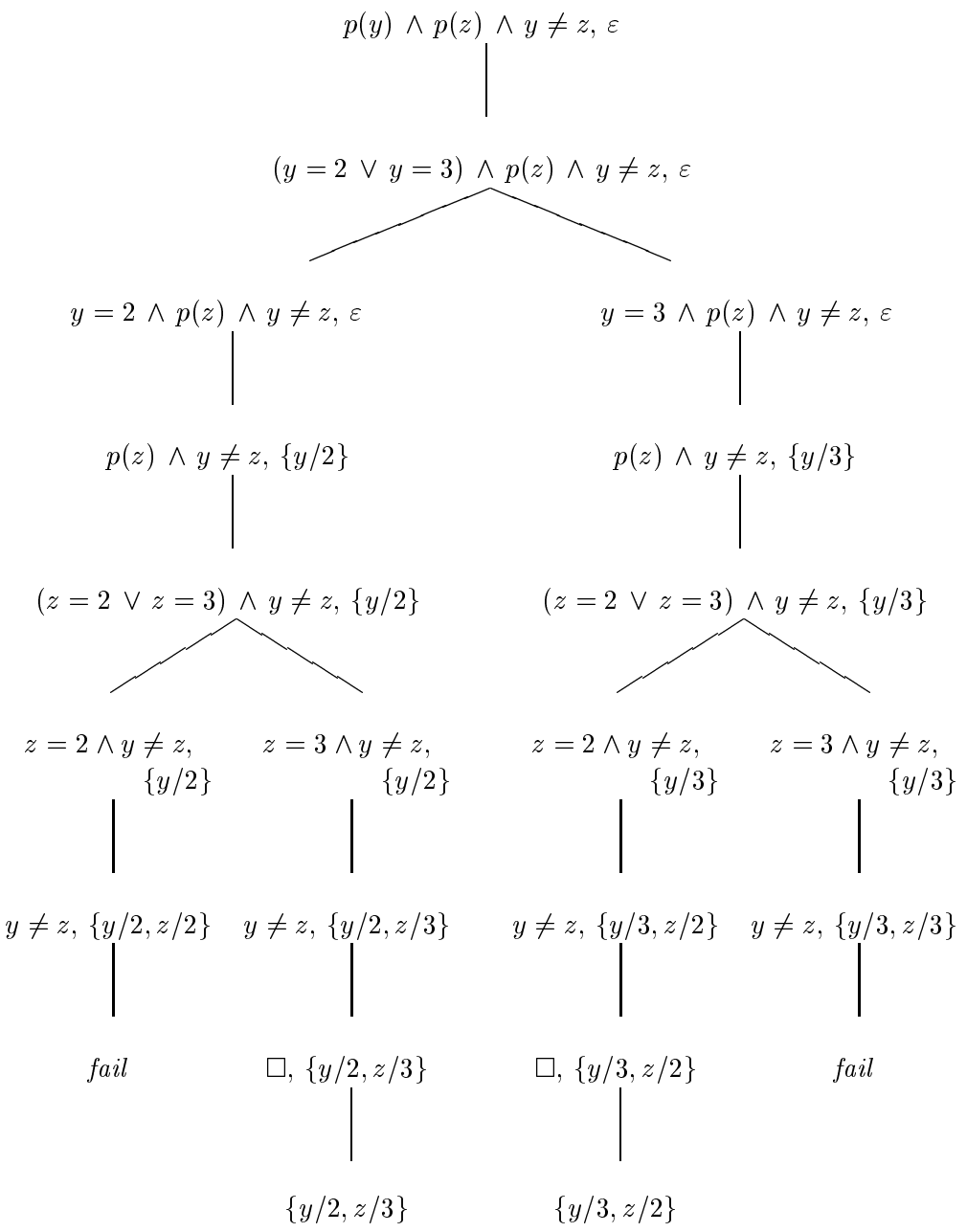
that is, the expression $\wp \rightarrow \phi^{\alpha}$ is true in the fixed interpretation.

For the success leaf $\alpha = \{y/2, z/3\}$ for example, this can be spelt out as follows: the formula

$$\forall x\,(p(x) \leftrightarrow x = 2 \vee x = 3) \rightarrow p(2) \wedge p(3) \wedge 2 \neq 3$$

is true, where we made explicit the fact that the $\wp$-axiom is universally closed w.r.t. x.

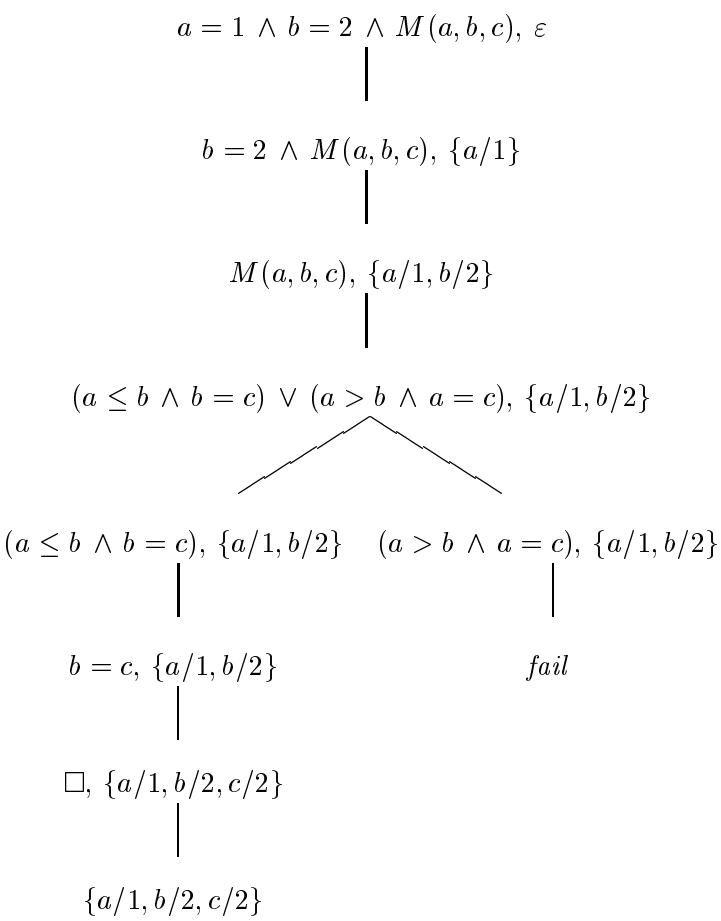
Likewise, for $\alpha = \{y/3, z/2\}$ we obtain an other true formula.



Example 2. The following implementation determines the maximum of two elements:

$$(M(x, y, z) \leftrightarrow (x \leq y \wedge y = z) \vee (x > y \wedge x = z)) \wedge a = 1 \wedge b = 2 \wedge M(a, b, c)$$

The computation tree for the formula, with the empty valuation ε is as follows:



The computation tree, though it has a failure leaf, is successful, the success leaf $\alpha = \{a/1, b/2, c/2\}$ yields values that make the formula $a = 1 \wedge b = 2 \wedge M(a,b,c)$ true relative to the $\wp$-axiom that defines M, that is:

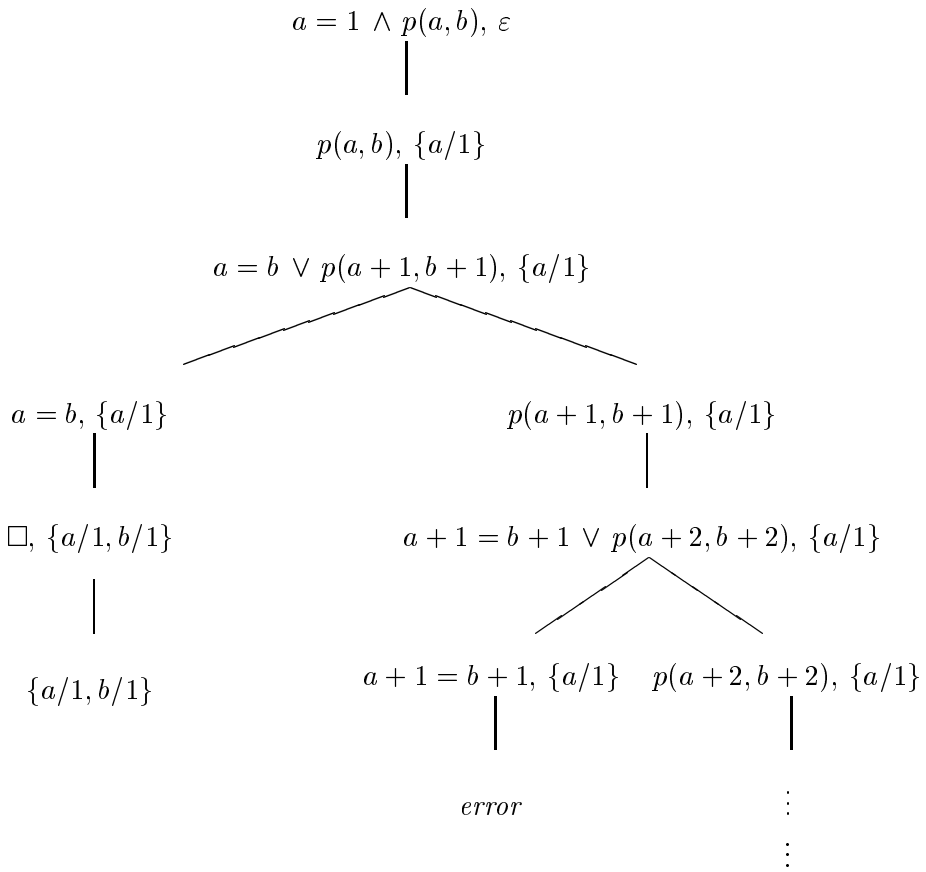
$$\forall\,(M(x,y,z) \leftrightarrow (x \leq y \wedge y = z) \vee (x > y \wedge x = z)) \rightarrow 1 = 1 \wedge 2 = 2 \wedge M(1,2,2)$$

is true.

Example 3. Let's consider now the following implementation: the $\wp$-axiom and ϕ are given as:

$$(p(x,y) \leftrightarrow x = y \vee p(x+1, y+1)) \wedge a = 1 \wedge p(a,b)$$

The computation tree for ϕ, with the empty valuation ε is as follows:



In this case, we have to deal with an infinite computation tree, that possesses a success leaf $\alpha = \{a/1, b/1\}$ which validates the formula $\wp \rightarrow \phi^{\alpha}$, that is,

$$\forall(p(x,y) \leftrightarrow x = y \vee p(x+1, y+1)) \rightarrow 1 = 1 \wedge p(1,1)$$

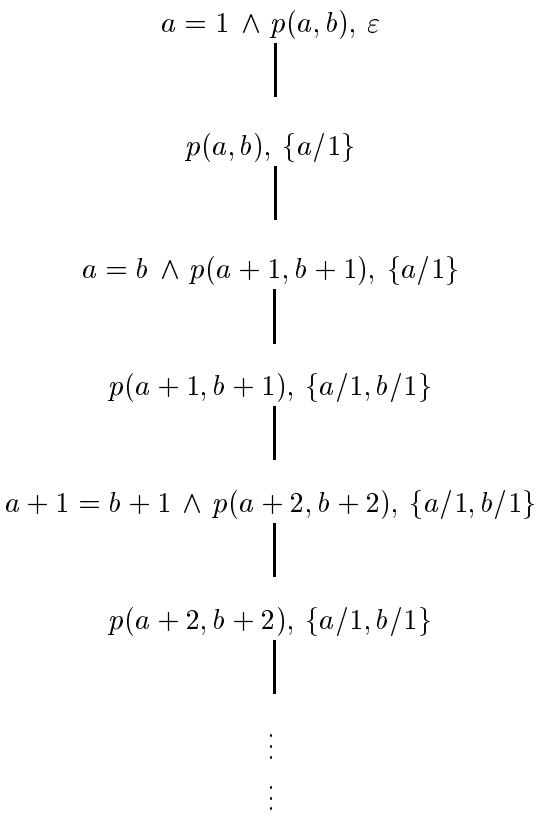
is true.

The computation tree, though infinite, has an error leaf, is determined and successful.

Example 4. In this case we consider an implementation with the same formula as in *example 3*, but here we associate to it a different $\wp$-axiom:

$$(p(x,y) \leftrightarrow x = y \wedge p(x+1, y+1)) \wedge a = 1 \wedge p(a,b)$$

The computation tree for the formula will be again infinite, but it will possess no labels at all, resulting an undetermined computation tree:



2.3 Soundness and Completeness

For a given implementation $\mathbf{I} = \wp \wedge \phi$, the computation mechanism defined in the previous section attempts to find a valuation α for the free variables of ϕ (possibly not for all its free variables) that makes the formula ϕ true relative to $\wp$, if the formula is satisfiable, in which case this can be expressed by the formalism:

$$\models \wp \rightarrow \forall(\phi^{\alpha})$$

or otherwise it reports a failure, that is $\wp \rightarrow \exists(\phi^{\alpha})$ is false.

It is worth noticing, that in case a $\wp$-axiom contains free variables of ϕ, then obviously instead of $\wp$ we should take in consideration $\wp^{\alpha}$, that is:

$$\models \wp^{\alpha} \rightarrow \forall(\phi^{\alpha})$$

This corresponds to implementations where procedures have direct access to global variables, a style usually avoided in programming, as procedures should communicate with the environment via their interface alone. Therefore we restrict our attention on cases when $\wp$-axioms do not access free variables of ϕ.

We start with a lemma which is helpful to keep track of valuations during a whole computation.

Lemma 2.3.1. For a given implementation $\mathbf{I} = \wp \wedge \phi$, and initial valuation α, $[\![\phi]\!]_\alpha$ contains only valuations extending α with pairs x/d or $a[t_1^\alpha, t_2^\alpha, ..., t_n^\alpha]/t^\alpha$, where x and array a occur free in ϕ, or appear existentially or universally bounded in ϕ, or further formulas that evolve from ϕ along the tree-path that leads to the extended valuation in case. Moreover, if ϕ is α-closed, then $[\![\phi]\!]_\alpha$ contains only valuations extending α with variables that appear existentially or universally quantified in ϕ or further formulas that evolve from ϕ along the tree-path that leads to the extended valuation in case.

Proof: By induction on the length of the path from the root to the extended valuation in case, based on the structure of the computation tree given as in Definition 5. □

The soundness of the computation mechanism is expressed by the following theorem.

Theorem 2.3.2. (Soundness) Let $\mathbf{I} = \wp \wedge \phi$ be an implementation, and α a valuation. Then:

1.) If $[\![\phi]\!]_\alpha$ contains a success leaf labelled with α', then α' extends α, and $\models \wp \rightarrow \forall(\phi^{\alpha'})$. (In particular, $\models \wp \rightarrow \exists(\phi^\alpha)$.)

2.) If $[\![\phi]\!]_\alpha$ is failed, then $\wp \rightarrow \exists(\phi^\alpha)$ is false.

The proof of this theorem is far from being obvious. Due to the fact that here we have to deal with possibly infinite computation trees, that may possibly have infinite subsidiary trees when negations or implications are present, and that the definition of the computation tree doesn't necessarily refer to lexicographically smaller formulas anymore, a simultaneous induction pattern for the proof of soundness similar to that of Apt and Bezem [1] cannot be applied directly. Therefore the proof for the general case is left as a further research topic. Basically, the problem arising here has a straightforward connection with SLDNF-resolution, where SLDNF-trees (finitely branching downward growing possibly infinite trees) are used to reason about soundness and completeness when negation is incorporated as finite failure rule. (See more in [5] and [3].)

The case studies of later chapters are based on formulas where the scope of negations and the premise part of implications consist of *non-procedural atoms*, that is, it is either of the form $s = t$, with s and t terms, or $p(t_1, t_2, ..., t_n)$ with p, an n-ary predefined relation symbol (and not one defined by a $\wp$-axiom).

The following lemma states soundness of the computation mechanism over non-procedural atoms.

Lemma 2.3.3. If A is a non-procedural atom, and α a valuation, then $[\![A]\!]_\alpha$ is a tree having as its only leaf either a success, a failure or an error leaf. Moreover:

1.) If it is a success leaf labelled with α', then α' extends α, such that A is α'-closed and $A^{\alpha'}$ is true. (In particular, $\exists(A^\alpha)$ is true in this case.)

2.) If it is a failure leaf, then A is α-closed and A^α is false.

Proof: We go through the cases of the clause 2. in Definition 5. of the computation tree, as this clause deals with atomic conjuncts. It is the case that the second conjunct ψ is the empty conjunction, $\psi = \Box$.

- Atom A has the form $s = t$, is α-closed and true, then A^α is true. The root of $[\![A]\!]_\alpha$ has $[\![\Box]\!]_\alpha$ as its only son, whose root, by clause 1. of the definition, has its only son the success leaf α.

- Atom A has the form $s = t$, is α-closed and false, then is A^α is false. The root of $[\![A]\!]_\alpha$ has the failure leave *fail* as its only son.

- Atom A has the form $s = t$, is not α-closed, but is not an α-assignment. Then the root of $[\![A]\!]_\alpha$ has the error leave *error* as its only son.

- Atom A is an α-assignment $s = t$, such that either s or t is a variable which is not α-closed, say $s \equiv x$, with x not α-closed and t α-closed. Then the root of $[\![A]\!]_\alpha$ has $[\![\Box]\!]_{\alpha'}$ as its subtree, where α' extends α with the pair x/t^α. According to this, A becomes α'-closed and $A^{\alpha'}$ is true, and after expanding $[\![\Box]\!]_{\alpha'}$, α' will become the only leaf of $[\![A]\!]_\alpha$. The symmetrical case is analogous.

- The case of the array element in an α-assignment follows the same pattern as the previous one.

- Atom A has the form $p(t_1, t_2, ..., t_n)$, where p is a predefined n-ary relation symbol, $t_1, t_2, ..., t_n$ are α-closed, and $p(t_1^\alpha, t_2^\alpha, ..., t_n^\alpha)$ is true, then A is α-closed and A^α is true. The root of $[\![A]\!]_\alpha$ has $[\![\Box]\!]_\alpha$ as its subtree, which after being expanded, will have the only success leaf α.

- Atom A has the form $p(t_1, t_2, ..., t_n)$, where p is a predefined n-ary relation symbol, $t_1, t_2, ..., t_n$ are α-closed, and $p(t_1^\alpha, t_2^\alpha, ..., t_n^\alpha)$ is false, then A is α-closed and obviously A^α is false. The root of $[\![A]\!]_\alpha$ has the failure leaf *failure* as its only son.

- Atom A has the form $p(t_1, t_2, ..., t_n)$, where p is a predefined n-ary relation symbol, and at least one of $t_1, t_2, ..., t_n$ is not α-closed. Then root of $[\![A]\!]_\alpha$ has the error leaf *error* as its only son.

- The last two cases of clause 2. of the definition consider procedural atomic formulas, so we don't need to deal with them in this proof.

Having considered all the possible cases for non-procedural atomic formulas, the proof of the lemma is complete now. $\Box$

Theorem 2.3.4. (Restricted Soundness) Let $\mathbf{I} = \wp \wedge \phi$ be an implementation, such that the scope of any negation and/or the premise part of any

implication consist of only one non-procedural atom, and let α be a valuation. Then:

1.) If $[\![\phi]\!]_\alpha$ contains a success leaf labelled with α', then α' extends α, and $\models \wp \to \forall(\phi^{\alpha'})$. (In particular, $\models \wp \to \exists(\phi^{\alpha})$.)

2.) If $[\![\phi]\!]_\alpha$ is failed, then $\wp \to \exists(\phi^{\alpha})$ is false.

Proof:

1.) We have seen in Lemma 2.3.1. that α' is an extension of α, moreover, the definition of the computation tree also ensures us, that α' is an extension of any intermediate valuation to which α has been extended along the path to the success leaf.

We use induction on the length of the path from the success leaf α' to the root, based on the structure of the computation tree given as in Definition 5.

We denote this length by N. One has to prove that for any k with $1 \leq k \leq N$:

$$\models \wp \to \forall(\phi_k^{\alpha'})$$

where ϕ_k denotes the formula at distance k from the success leaf α' along the path up to the root. In particular, $\phi_N = \phi$.

Base case: $k = 1$. According to the structure of a computation tree, there may be two cases:

- ϕ_1 is the empty conjunction, in which case $\models \wp \to \forall(\phi_1^{\alpha'})$ is trivial, or
- ϕ_1 is of the form $\forall x \in [s..t]\ \psi(x)$ with $s^{\alpha'} > t^{\alpha'}$. Then $\forall x \in \left[s^{\alpha'}..t^{\alpha'}\right]\ \psi^{\alpha'}(x)$ is trivially true, which results $\models \wp \to \forall(\phi_1^{\alpha'})$.

Thus, the base case holds. For the induction step, assume:

Hypothesis: We presume, that for $k = i$ with $i < N$ we have:

$$\models \wp \to \forall(\phi_i^{\alpha'})$$

We have to prove: $\models \wp \to \forall(\phi_{i+1}^{\alpha'})$

We follow the structure of the computation tree given as in Definition 5, excluding, of course, all cases when the subtrees are leaves (failure or error leaves), because it is the case that ϕ_i is a formula that corresponds to an internal node of the tree along the success path from the success leaf towards the root. (In particular, with the cases of success leaves we have dealt in the base step.)

When necessary, we refer to the valuation at the node corresponding to ϕ_{i+1} by β.

- Clause 2
  - $\phi_{i+1} = A \wedge \phi_i$, where A is of the form $s = t$, is β-closed and true. As we have $\beta \subseteq \alpha'$, A is α'-closed and $A^{\alpha'}$ is true as well. By this and by hypothesis we conclude $\wp \to \forall(\phi_{i+1}^{\alpha'})$ is true.

- $\phi_{i+1} = A \wedge \phi_i$, where A is a β-assignment $x = t$, then β is extended with the pair x/t^β, and as $\beta \cup \{x/t^\beta\} \subseteq \alpha'$, we have that: $A^{\alpha'}$ is true. By this and by hypothesis we conclude: $\wp \rightarrow \forall(\phi_{i+1}^{\alpha'})$ is true.
- By analogous reasoning for the case $\phi_{i+1} = A \wedge \phi_i$, where A is a β-assignment $a[t_1, t_2, ..., t_n] = t$, we obtain $\wp \rightarrow \forall(\phi_{i+1}^{\alpha'})$ is true.
- $\phi_{i+1} = p(t_1, t_2, ..., t_n) \wedge \psi$ and $\phi_i = \psi$, with p predefined n-ary relation symbol, and $p(t_1^\beta, t_2^\beta, ..., t_n^\beta)$ true. Then $(p(t_1, t_2, ..., t_n))^\beta$ is true as well, and as $\beta \subseteq \alpha'$, we have $(p(t_1, t_2, ..., t_n))^{\alpha'}$ true, which together with the hypothesis results: $\wp \rightarrow \forall(\phi_{i+1}^{\alpha'})$ is true.
- $\phi_{i+1} = p(t_1, t_2, ..., t_n) \wedge \psi_1$ and $\phi_i = \psi\{x_1/t_1, ..., x_n/t_n\} \wedge \psi_1$. Then by the equivalence $p(x_1, x_2, ..., x_n) \leftrightarrow \psi$ from $\wp$, we obtain (we call again the reader's attention that we requested ψ not to contain free variables of ϕ):
  $(\psi\{x_1/t_1, ..., x_n/t_n\})^{\alpha'} \equiv \psi\{x_1/t_1^{\alpha'}, ..., x_n/t_n^{\alpha'}\} \leftrightarrow$
  $\leftrightarrow p(t_1^{\alpha'}, t_2^{\alpha'}, ..., t_n^{\alpha'}) \equiv (p(t_1, t_2, ..., t_n))^{\alpha'}$.
  Which by hypothesis results: $\wp \rightarrow \forall(\phi_{i+1}^{\alpha'})$ is true.

- Clause 3: $\phi_{i+1} = (\psi_1 \vee \psi_2) \wedge \psi$ and $\phi_i = \psi_2 \wedge \psi$. Then, by the distributive law: $(\psi_1 \vee \psi_2) \wedge \psi \equiv (\psi_1 \wedge \psi) \vee (\psi_2 \wedge \psi)$ and hypothesis, we conclude that$\wp \rightarrow \forall(\phi_{i+1}^{\alpha'})$ is true as well.

- Clause 4: $\phi_{i+1} = (\psi_1 \wedge \psi_2) \wedge \psi$ and $\phi_i = \psi_1 \wedge (\psi_2 \wedge \psi)$. It is straightforward, by the associativity of conjunction and hypothesis, that $\wp \rightarrow \forall(\phi_{i+1}^{\alpha'})$ is true as well

- Clause 5: We distinguish two cases:

  - $\phi_{i+1} = (\psi_1 \rightarrow \psi_2) \wedge \psi$ and $\phi_i = \psi$, where ψ_1 is β-closed and $[\![\psi_1]\!]_\beta$ is finite and contains only failure leaves. Then under the assumption of the theorem, it is the case that ψ_1 is a non-procedural atom. By Lemma 2.3.3. ψ_1^β is false, in particular, as ψ_1 is β-closed and $\beta \subseteq \alpha'$: ψ_1 is α'-closed and false, which means that $\neg\psi_1^{\alpha'}$ is true. Then $\wp \rightarrow \forall(\neg\psi_1^{\alpha'} \vee \psi_2^{\alpha'})$ is true as well. From here, by the logical equivalence of $\psi_1 \rightarrow \psi_2$ and $\neg\psi_1 \vee \psi_2$, and by hypothesis we can conclude: $\wp \rightarrow \forall(\phi_{i+1}^{\alpha'})$ is true as well.
  - $\phi_{i+1} = (\psi_1 \rightarrow \psi_2) \wedge \psi$ and $\phi_i = \psi_2 \wedge \psi$, where ψ_1 is β-closed and $[\![\psi_1]\!]_\beta$ contains a success leaf. Then under the assumption of the theorem, it is the case that ψ_1 is a non-procedural atom. By Lemma 2.3.3. ψ_1^β is true, in particular, as ψ_1 is β-closed and $\beta \subseteq \alpha'$: ψ_1 is α'-closed and $\psi_1^{\alpha'}$ is true, which means that $\neg\psi_1^{\alpha'}$ is false. Then we apply the logical equivalence of $\psi_1 \rightarrow \psi_2$ and $\neg\psi_1 \vee \psi_2$ and we have that: $(\psi_1 \rightarrow \psi_2)^{\alpha'} \equiv (\neg\psi_1 \vee \psi_2)^{\alpha'} \equiv \neg\psi_1^{\alpha'} \vee \psi_2^{\alpha'} \equiv \psi_2^{\alpha'}$. This by hypothesis results: $\wp \rightarrow \forall(\phi_{i+1}^{\alpha'})$ is true as well.

- Clause 6: $\phi_{i+1} = \neg\psi_1 \wedge \psi$ and $\phi_i = \psi$, where ψ_1 is β-closed and $[\![\psi_1]\!]_\beta$ is finite and contains only failure leaves. Then under the assumption of the theorem, it is the case that ψ_1 is a non-procedural atom. By Lemma 2.3.3. ψ_1^β is false, in particular, as ψ_1 is β-closed and $\beta \subseteq \alpha'$: ψ_1 is α'-closed $\psi_1^{\alpha'}$ is false, which means that $\neg\psi_1^{\alpha'}$ is true. This and the hypothesis: $\models \wp \to \forall(\psi^{\alpha'})$, results that $\wp \to \forall(\phi_{i+1}^{\alpha'})$ is true.

- Clause 7: $\phi_{i+1} = \exists x\ \psi_1 \wedge \psi$ and $\phi_i = \psi_1 \wedge \psi$, under the assumption that x is fresh w.r.t. β and ψ. It is convenient to make the possible occurrence of x in ψ_1 explicit by writing $\psi_1(x)$ for ψ_1. By hypothesis $\wp \to \forall(\psi_1(x) \wedge \psi)^{\alpha'}$ is true, it follows that $\wp \to \forall(\exists x\, \psi_1(x) \wedge \psi)^{\alpha'}$ is true as well, even in case, when x doesn't occur in the domain of α', in which case we can apply the logical consequence $\forall x\, \psi_!(x) \to \exists x\, \psi_1(x)$.

- Clause 8: $\phi_{i+1} = \exists x\ \in [s..t]\, \psi(x) \wedge \psi_1$ and
  $\phi_i = (\psi(x) \vee \exists y \in [s+1\,..t]\psi(y)) \wedge \psi_1$, where β is extended with the pair $\{x/s^\beta\}$, but as $\beta \subseteq \alpha'$, we may also write $\{x/s^{\alpha'}\}$. We have the following equivalences:
  $((\psi(x) \vee \exists y \in [s+1\,..t]\psi(y)) \wedge \psi_1)^{\alpha'} \equiv$
  $\left(\psi^{\alpha'}(x^{\alpha'}) \vee \exists y \in [s^{\alpha'}+1\,..t^{\alpha'}]\psi^{\alpha'}(y)\right) \wedge \psi_1^{\alpha'} \equiv$
  $\left(\psi^{\alpha'}(s^{\alpha'}) \vee \exists y \in [s^{\alpha'}+1\,..t^{\alpha'}]\psi^{\alpha'}(y)\right) \wedge \psi_1^{\alpha'} \equiv$
  $\exists y \in [s^{\alpha'}..t^{\alpha'}]\psi^{\alpha'}(y) \wedge \psi_1^{\alpha'} \equiv$
  $(\exists y \in [s..t]\psi(y) \wedge \psi_1)^{\alpha'}$
  Which by hypothesis results: $\wp \to \forall(\phi_{i+1}^{\alpha'})$ is true as well.
  Here we require the usual care with bound variables, as y may not be fresh w.r.t. α', but it is for β, and so is x, so substitution may be applied.

- Clause 9: $\phi_{i+1} = \forall x\ \in [s..t]\, \psi(x) \wedge \psi_1$ and $\phi_i = (\psi(x) \wedge \forall y\ \in [s+1\,..t]\, \psi(y)) \wedge \psi_1$, where β is extended with the pair $\{x/s^\beta\}$, but as $\beta \subseteq \alpha'$, we may write $\{x/s^{\alpha'}\}$. Then we have:
  $((\psi(x) \wedge \forall y \in [s+1\,..t]\psi(y)) \wedge \psi_1)^{\alpha'} \equiv$
  $\left(\psi^{\alpha'}(x^{\alpha'}) \wedge \forall y \in [s^{\alpha'}+1\,..t^{\alpha'}]\psi^{\alpha'}(y)\right) \wedge \psi_1^{\alpha'} \equiv$
  $\left(\psi^{\alpha'}(s^{\alpha'}) \wedge \forall y \in [s^{\alpha'}+1\,..t^{\alpha'}]\psi^{\alpha'}(y)\right) \wedge \psi_1^{\alpha'} \equiv$
  $\forall y \in [s^{\alpha'}..t^{\alpha'}]\psi^{\alpha'}(y) \wedge \psi_1^{\alpha'} \equiv$
  $(\forall y \in [s..t]\psi(y) \wedge \psi_1)^{\alpha'}$
  Which by hypothesis results: $\wp \to \forall(\phi_{i+1}^{\alpha'})$ is true as well.

The induction step holds. Thus the property holds for any k with $1 \leq k \leq N$, in particular for $k = N$, then we have:

$$\models \wp \to \forall(\phi^{\alpha'})$$

2.) By induction on the structure of the computation tree, for which it is the case that it is finite and it contains only failure leaves.

Base case: this corresponds to those subtrees, for which the root itself has a failure leaf as its only son. According to the definition of the tree there may be four such cases (we refer by β to the valuation at the root of such a subtree):

- the root has the form: $A \wedge \psi$, β with atom A of the form $s = t$, that is β-closed and false, that is A^β is false. This obviously results, that $\wp \rightarrow \exists(A \wedge \psi)^\beta$ is false as well.

- the root has the form: $p(t_1, t_2, ..., t_n) \wedge \psi$, β, where p is a predefined n-ary relation symbol, and and $p(t_1^\beta, t_2^\beta, ..., t_n^\beta)$ is false. Then $p^\beta(t_1, t_2, ..., t_n)$ is false as well, which results that $\wp \rightarrow \exists(p(t_1, t_2, ..., t_n) \wedge \psi)^\beta$ is false.

- the root has the form: $\neg\psi_1 \wedge \psi$, β, where $[\![\psi_1]\!]_\beta$ contains at least one success leaf. Then by the assumption of the theorem, it is the case, that ψ_1 is a non-procedural atom, so Lemma 2.3.3. applies, that is $\forall(\psi_1^\beta)$ is true. From here we conclude, that $\exists(\neg\psi_1^\beta)$ is false, and then $\wp \rightarrow \exists(\neg\psi_1 \wedge \psi)^\beta$ is false as well.

- the root has the form: $\exists x \in [s..t]\psi(x) \wedge \psi_1$, β, where $s^\beta > t^\beta$. Then $\exists x \in [s^\beta..t^\beta]\psi^\beta(x)$ is trivially false, and so is:$\wp \rightarrow \exists(\exists x \in [s..t]\psi(x) \wedge \psi)^\beta$.

Induction step: This step consists of proving for any node of the tree, say labelled by ψ, β, that $\wp \rightarrow \exists(\psi)^\beta$ is false, by applying the induction hypothesis to every subtree of this node, $[\![\psi_1]\!]_{\beta'}$ $(\beta \subseteq \beta')$, that is $\wp \rightarrow \exists(\psi_1)^{\beta'}$ is false. (These subtrees are of course finite, containing only failure leaves, so the induction hypothesis may be applied to them).

We again follow the structure of the computation tree given as in Definition 5., excluding of course all cases when the subtree would be an error or a success leaf. We also call the reader's attention, that with the cases of failure leaves we have dealt in the base case.

- Clause 2

  - $\psi = A \wedge \psi_1$, where A is of the form $s = t$, is β-closed and true. By induction hypothesis, $\wp \rightarrow \exists(\psi_1)^\beta$ is false, which yields $\wp \rightarrow \exists(A \wedge \psi_1)^\beta$ is false, even if A^β is true.
  - $\psi = A \wedge \psi_1$, where A is a β-assignment $x = t$, then β is extended with the pair x/t^β, s.t. $\beta' = \beta \cup \{x/t^\beta\}$. By induction hypothesis, $\wp \rightarrow \exists(\psi_1)^{\beta'}$ is false. If x doesn't occur free in ψ_1, then it is straightforward, that $\wp \rightarrow \exists(\psi_1)^\beta$ is false as well, and so is$\wp \rightarrow \exists(A \wedge \psi_1)^\beta$. If x occur free in ψ_1, it is the case, that the β-assignment, resulting $\{x/t^\beta\}$ doesn't validate $\exists(\psi_1)^\beta$, and then it wouldn't validate $A \wedge (\psi_1)^\beta$, and so $\wp \rightarrow \exists(A \wedge \psi_1)^\beta$ is false as well.

  - By analogous reasoning for the case $\psi = A \wedge \psi_1$, where A is a β-assignment $a[t_1, t_2, ..., t_n] = t$, we obtain $\wp \rightarrow \exists(A \wedge \psi_1)^\beta$ is false as well.
  - $\psi = p(t_1, t_2, ..., t_n) \wedge \psi_1$, with p predefined n-ary relation symbol. By hypothesis, $\wp \rightarrow \exists(\psi_1)^\beta$ is false, which yields $\wp \rightarrow \exists(p(t_1, t_2, ..., t_n) \wedge \psi_1)^\beta$ is false as well, even if $p(t_1^\beta, t_2^\beta, ..., t_n^\beta)$ is true.
  - $\psi = p(t_1, t_2, ..., t_n) \wedge \psi_1$, with p, an n-ary relation symbol defined by the equivalence $p(x_1, x_2, ..., x_n) \leftrightarrow \psi$ from $\wp$. We obtain (we call again the reader's attention that we requested ψ not to contain free variables of ϕ):
    $(\psi \{x_1/t_1, ..., x_n/t_n\})^\beta \equiv \psi \left\{x_1/t_1^\beta, ..., x_n/t_n^\beta\right\} \leftrightarrow$
    $\leftrightarrow p(t_1^\beta, t_2^\beta, ..., t_n^\beta) \equiv (p(t_1, t_2, ..., t_n))^\beta$.
    Which by hypothesis results: $\wp \rightarrow \exists(p(t_1, t_2, ..., t_n) \wedge \psi_1)^\beta$ is false as well.

- Clause 3: $\psi = (\psi_1 \vee \psi_2) \wedge \psi_3$. Applying the hypothesis to the subtrees, we obtain: $\wp \rightarrow \exists(\psi_1 \wedge \psi_3)^\beta$ and $\wp \rightarrow \exists(\psi_2 \wedge \psi_3)^\beta$ are false. Then, by the distributive law: $(\psi_1 \vee \psi_2) \wedge \psi_3 \equiv (\psi_1 \wedge \psi_3) \vee (\psi_2 \wedge \psi_3)$, we conclude that $\wp \rightarrow \exists((\psi_1 \vee \psi_2) \wedge \psi_3)^\beta$ is false as well.

- Clause 4: $\psi = (\psi_1 \wedge \psi_2) \wedge \psi_3$. It is straightforward, by the associativity of conjunction and applying the hypothesis $\wp \rightarrow \exists(\psi_1 \wedge (\psi_2 \wedge \psi_3))^\beta$ is false, that $\wp \rightarrow \exists((\psi_1 \wedge \psi_2) \wedge \psi_3)^\beta$ will be false as well.

- Clause 5: We distinguish two cases:

  - $\psi = (\psi_1 \rightarrow \psi_2) \wedge \psi_3$. By induction hypothesis, $\wp \rightarrow \exists(\psi_3)^\beta$ is false, then so is: $\wp \rightarrow \exists((\psi_1 \rightarrow \psi_2) \wedge \psi_3)^\beta$.
  - $\psi = (\psi_1 \rightarrow \psi_2) \wedge \psi_3$. By induction hypothesis, $\wp \rightarrow \exists(\psi_2 \wedge \psi_3)^\beta$ is false. It is also the case, that ψ_1 is a β-closed non-procedural atom. Lemma 2.3.3. is applied: ψ_1^β is true, which means that $\neg\psi_1^{\alpha'}$ is false. Then we apply the logical equivalence of $\psi_1 \rightarrow \psi_2$ and $\neg\psi_1 \vee \psi_2$ and we have that: $(\psi_1 \rightarrow \psi_2)^\beta \equiv (\neg\psi_1 \vee \psi_2)^\beta \equiv \neg\psi_1^\beta \vee \psi_2^\beta \equiv \psi_2^\beta$. This, by hypothesis results: $\wp \rightarrow \exists((\psi_1 \rightarrow \psi_2) \wedge \psi_3)^\beta$ is false.

- Clause 6: $\psi = \neg\psi_1 \wedge \psi_2$, and by hypothesis $\wp \rightarrow \exists(\psi_2)^\beta$ is false, then so is: $\wp \rightarrow \exists(\neg\psi_1 \wedge \psi_2)^\beta$.

- Clause 7: $\psi = \exists x\ \psi_1(x) \wedge \psi_2$. By hypothesis, $\wp \rightarrow \exists(\psi_1(x) \wedge \psi_2)^\beta$ is false. Then it is straightforward, that $\wp \rightarrow \exists(\exists x\, \psi_1(x) \wedge \psi_2)^\beta$ is false as well.

- Clause 8: $\psi = \exists x \in [s..t]\, \psi_1(x) \wedge \psi_2$, with x fresh w.r.t β, and by hypothesis $\wp \rightarrow \exists\left((\psi_1(x) \vee \exists y \in [s+1\,..t]\psi_1(y)) \wedge \psi_2\right)^{\beta'}$ is false, where

$\beta' = \beta \cup \{x/s^{\beta}\}$, and y is fresh w.r.t. β'. We have the following equivalences:
$((\psi_1(x) \vee \exists y \in [s+1\,..t]\psi_1(y)) \wedge \psi_2)^{\beta'} \equiv$
$\left(\psi_1^{\beta'}(x^{\beta'}) \vee \exists y \in [s^{\beta'}+1\,..t^{\beta'}]\psi_1^{\beta'}(y)\right) \wedge \psi_2^{\beta'} \equiv$
$\left(\psi_1^{\beta'}(s^{\beta'}) \vee \exists y \in [s^{\beta'}+1\,..t^{\beta'}]\psi_1^{\beta'}(y)\right) \wedge \psi_2^{\beta'} \equiv$
$\exists y \in [s^{\beta'}..t^{\beta'}]\psi_1^{\beta'}(y) \wedge \psi_2^{\beta'} \equiv$
$(\exists y \in [s..t]\psi_1(y) \wedge \psi_2)^{\beta'}$
Which by hypothesis results: $\wp \rightarrow \exists\,(\exists y \in [s..t]\psi_1(y) \wedge \psi_2)^{\beta'}$ is false as well. As y is fresh w.r.t. β', we also have: $\wp \rightarrow \exists\,(\exists y \in [s..t]\psi_1(y) \wedge \psi_2)^{\beta}$ is false, and as x is fresh w.r.t. β, we conclude:
$\wp \rightarrow \exists\,(\exists x \in [s..t]\psi_1(x) \wedge \psi_2)^{\beta}$ is false.

- Clause 9: $\psi = \forall x \in [s..t]\,\psi_1(x) \wedge \psi_2$, with x fresh w.r.t β, and by hypothesis $\wp \rightarrow \exists\,((\psi_1(x) \vee \exists y \in [s+1\,..t]\psi_1(y)) \wedge \psi_2)^{\beta'}$ is false, where $\beta' = \beta \cup \{x/s^{\beta}\}$, and y is fresh w.r.t. β'. We have the following equivalences:
$((\psi_1(x) \wedge \forall y \in [s+1\,..t]\psi_1(y)) \wedge \psi_2)^{\beta'} \equiv$
$\left(\psi_1^{\beta'}(x^{\beta'}) \wedge \forall y \in [s^{\beta'}+1\,..t^{\beta'}]\psi_1^{\beta'}(y)\right) \wedge \psi_2^{\beta'} \equiv$
$\left(\psi_1^{\beta'}(s^{\beta'}) \wedge \forall y \in [s^{\beta'}+1\,..t^{\beta'}]\psi_1^{\beta'}(y)\right) \wedge \psi_2^{\beta'} \equiv$
$\forall y \in [s^{\beta'}..t^{\beta'}]\psi_1^{\beta'}(y) \wedge \psi_2^{\beta'} \equiv$
$(\forall y \in [s..t]\psi_1(y) \wedge \psi_2)^{\beta'}$
Which by hypothesis results: $\wp \rightarrow \exists\,(\forall y \in [s..t]\psi_1(y) \wedge \psi_2)^{\beta'}$ is false as well. As y is fresh w.r.t. β', we also have: $\wp \rightarrow \exists\,(\forall y \in [s..t]\psi_1(y) \wedge \psi_2)^{\beta}$ is false, and as x is fresh w.r.t. β, we conclude:
$\wp \rightarrow \exists\,(\forall x \in [s..t]\psi_1(x) \wedge \psi_2)^{\beta}$ is false.

The induction step holds, thus the property is true for any node in the tree, in particular, for the root itself: ϕ, α, that is $\wp \rightarrow \exists(\phi^{\alpha})$ is false.□

The computation mechanism defined in Section 2.1 is obviously incomplete due to the possibility of errors. The following results state, that in the absence of errors, and with the restriction made on negation and implication, the computation mechanism is complete.

Theorem 2.3.5. (Restricted Completeness) Let $\mathbf{I} = \wp \wedge \phi$ be an implementation, such that the scope of any negation and/or the premise part of any implication consist of only one non-procedural atom, and let α be a valuation, such that $[\![\phi]\!]_{\alpha}$ is determined. Then:

1.) Suppose $\wp \rightarrow \exists(\phi^{\alpha})$ is true. Then the tree $[\![\phi]\!]_{\alpha}$ is successful.

2.) Suppose $\wp \rightarrow \exists(\phi^{\alpha})$ is false. Then the tree $[\![\phi]\!]_{\alpha}$ is failed.

Proof: The proof is by reductio ad absurdum, using the results of the restricted soundness theorem, and basically it follows the proof pattern of the

restricted completeness theorem provided in Apt and Bezem [1].

2.4 Soundness Results Applied to Specifications

First, we show through a simple example how these soundness results can be applied to specifications.

Consider the problem of finding the maximum of two elements. This problem may be specified as follows:

$$\begin{aligned}\mathbf{S}(a,b,c) \quad =_{def} \quad & c \text{ is the maximum of } a \text{ and } b \Leftrightarrow \\ & (c = a \vee c = b) \wedge c \geq a \wedge c \geq b\end{aligned}$$

The following Alma-0 program represents an implementation for this problem for given a and b input data (see more on Alma-0 in Chapter 3):

```
VAR
  a,b,c :  INTEGER;

PROCEDURE Max (MIX x,y,z:  INTEGER);
BEGIN
  IF x <= y THEN y = z
            ELSE x = z
  END;
END Max;

BEGIN
  a = 1;
  b = 2;
  Max(a,b,c);
END;
```

This program segment has a declarative semantics, consequently it has a dual reading as a formula. In Section 2.2, the implementation of Example 2. (page 24) constitutes the translation of this program segment into a formula in the fixed interpretation. (In preliminary the reader may consult the *Translation Process Table* of Alma-0 in Section 3.2, on page 42.) We recall this formula:

$$\begin{aligned}\mathbf{I}(a,b,c) =_{def} & (M(x,y,z) \leftrightarrow (x \leq y \wedge y = z) \vee (x > y \wedge x = z)) \wedge \\ & a = 1 \wedge b = 2 \wedge M(a,b,c)\end{aligned}$$

We have seen that the computation mechanism yielded a success leaf $\alpha = \{a/1, b/2, c/2\}$ which validated the formula above, or expressed in the defined formalism:

$$\forall\,(M(x,y,z) \leftrightarrow (x \leq y \wedge y = z) \vee (x > y \wedge x = z)) \rightarrow 1 = 1 \wedge 2 = 2 \wedge M(1,2,2)$$

is true, this being a straightforward result of the restricted soundness theorem.

This means, that in case the following implication is true:

$$\forall a, b, c\ ((\mathbf{I}(a, b, c) \to \mathbf{S}(a, b, c))$$

the valuation $\alpha = \{a/1, b/2, c/2\}$ validates the specification as well, that is, our computational problem for the given input data is solved, the output data being delivered in c. We disregard now from a detailed proof of this implication, this relying only on basic logical reasoning. (Proofs of such implications for complex cases may be found in the case studies of Chapters 5–7.)

In general, we may formalise this approach as follows: given a specification $\mathbf{S}$, let $\mathbf{I} = \wp \wedge \phi$ be a declarative implementation for $\mathbf{S}$ in the fixed interpretation, such that $\mathbf{I}$ also satisfies the requirements of the restricted soundness theorem. Then applying the restricted soundness theorem, we obtain that the implementation $\mathbf{I}$ is sound, that is, if α' is a success leaf delivered by the computation mechanism (e.g. result obtained by running the corresponding program), then

$$\models \wp \to \forall(\phi^{\alpha'}). \tag{2.2}$$

If we can prove by some formal logic reasoning:

$$\models \forall(\wp \wedge \phi \to \mathbf{S}) \tag{2.3}$$

then (2.2) and (2.3) imply:

$$\models \wp \to \forall(\mathbf{S}^{\alpha'})$$

that is

$$\models \forall(\mathbf{S}^{\alpha'})$$

as the specification itself doesn't depend on the $\wp$-axioms, that is, α' validates the specification.

Therefore, in each case taken in consideration, our only task remains to prove the implication $\forall(\mathbf{I} \to \mathbf{S})$.

Note: This approach of reasoning about the correctness of implementations is to some extent the counterpart of the approach "correct-by-construction" program design, which is the discipline of calculating programs from their specifications, where correctness of an implementation is assured by self-evident steps along which the implementation has been developed. (See more in [4].)

Chapter 3

Alma-0 Programming Language

3.1 General Aspects of the Language

The Alma-0 programming language is an implemented programming language that supports declarative programming, and which combines the advantages of logic and imperative programming.

Basically, the language is an extension of a subset of Modula-2 that includes several new features inspired by the logic programming paradigm.

The implementation of the language is based on the Alma Abstract Architecture (AAA), a virtual architecture used during the intermediate code generation phase for the Alma-0 compiler. This combines the features of a RISC architecture and the WAM abstract machine [13]. AAA entails translating the AAA instructions into C statements, but its design is such, that it should be possible to translate them into machine code. A complete presentation of the implementation can be found in [9].

We recall in this chapter only a few of the new language features, relevant for declarative semantics, some of them being used in the case studies of Chapters 5–7 and refer to [2] for a detailed presentation of the language.

- Boolean expressions can be used as statements. If an expression evaluates to `TRUE`, the execution continues after the statement; if it evaluates to `FALSE`, we say that the statement *fails*, or a *failure* has occurred; and if no failure occurs during the execution of a sequence of statements, we say that the sequence of statements *succeeds*, otherwise it fails.

- Statements can be used as boolean expressions. If the sequence of statements succeeds, the expression evaluates to `TRUE`; if it fails, the expression evaluates to `FALSE`.

- Choice points can be created by the non-deterministic statements `ORELSE` and `SOME`. The former is a dual of the statement composition and the latter is a dual of the `FOR` statement. Upon failure the control returns to the most recent choice point, possibly within a procedure body, and

the computation resumes with the next branch in the state in which the previous branch was entered.

- The notion of *initialised* variable is introduced and the equality test is generalised to an assignment statement in case one side is an uninitialised variable, and the other side is an expression with known value. For example, the behaviour of the comparison `s = t` depends upon `s` and `t`:
  - if `s` and `t` are expressions with known values, a regular comparison is performed;
  - if `s` is uninitialised, and `t` is an expression with a known value, the value of `t` is assigned to `s`;
  - if `t` is uninitialised, and `s` is an expression with a known value, the value of `s` is assigned to t;
  - All remaining cases generate a run-time error.

- A new parameter-passing mechanism is introduced for variables of simple types, that makes it possible to use a procedure both for testing and computing. We call this parameter mechanism *call by mixed form*, and denote its use by the keyword `MIX`. Assume that the formal parameter is of a simple type. Then:
  - if the actual parameter is a variable, then it is passed by variable
  - if the actual parameter is an expression that is not a variable, its value is computed and assigned to a new variable v (generated by the compiler): it is v that is then passed by variable. So in this case the call by mixed form boils down to call by value.

  Using this parameter mechanism we can pass both expressions with known values and uninitialised variables as actual parameters. (Note: if the actual parameter is an expression that is not a variable, but contains an uninitialised variable, that is, the expression is without a known value, a run-time error occurs.)

The Alma-0 implementation does not realise faithfully the computation mechanism of section 2.1 in the sense that it may miss solutions, and it may proceed over cases where the computation mechanism reports an error.

Regarding the first case, there exists formulas for which the computation tree yields success leaves whereas the execution of the corresponding Alma-0 program gives a run-time error. In this respect, we can say that Alma-0 is incomplete. Consider the following examples:

- the computation tree of the formula $x = y \vee x = 0$ for the empty valuation, besides an error leaf, will also yield a success leaf $\{x/0\}$, that validates the formula, whereas the corresponding Alma-0 sequence (see next section): `EITHER x=y ORELSE x=0 END` gives a run-time error.

- the formula $p(y{+}1)$, where p is defined by the $\wp$-axiom: $p(x) \leftrightarrow (\mathrm{T} \vee x = 1)$. The computation tree, besides an error leaf, will also yield the empty valuation as success leaf, which shouldn't come as a surprise, as $p(y{+}1)$ is true independent of y. Unfortunately, the execution of the corresponding Alma-0 program gives a run-time error.

As long as errors are concerned, on one hand, as the use of insufficiently instantiated atoms in Alma-0 programs is to be discouraged, the Alma-0 implementation follows the computation mechanism of section 2.1 in cases: an evaluation of an atom that is not α-closed but is not an α-assignment yields a run-time error, and so do: the evaluation of an atom consisting of a predefined relation symbol with at least one not α-closed argument, and the cases of existentially or universally bounded formulas with not α-closed boundaries. The evaluation of an atom consisting of a relation symbol that is not predefined nor there exists a $\wp$-axiom for it, a compiling error arises.

On the other hand, according to [1], in the remaining two cases when the evaluation (of the computation mechanism) ends with the *error* leaf, in the cases of negation and implication, the computation process of Alma-0 simply proceeds. This is not always the case. Consider the following example: `NOT(x=y); x=0; y=1`. Ideally, the whole computation should succeed, but the execution of the program gives a run-time error, and so does the computation mechanism, so in this case Alma-0 follows the computation mechanism. But in the following case Alma-0 indeed diverges from the computation mechanism: `NOT(x=0); x=0`. The computation mechanism reports an error (the scope of the negation is not α-closed), whereas the execution of the corresponding Alma-0 program fails, which is correct. Yet, Alma-0 doesn't follow exactly the same compromise here as the implementations of Prolog, unlike [1] suggests. For example in case `NOT(NOT(x=0)); x=1`, Alma-0 fails whereas Prolog, wrongly, succeeds. In this respect we can say, that Alma-0 is sound, unlike Prolog.

3.2 Declarative Interpretation

Alma-0 has been designed with the view of promoting declarative programming. Recall again that in our context we consider a program declarative if its meaning can be described by means of a logical formula that can be obtained by means of a syntax directed translation. We call then this formula the *declarative interpretation* of the program.

By assigning to this formula its semantic meaning that agrees with the operational semantics of the original program we obtain *declarative semantics* of the program under consideration. Alma-0 programs built out of a limited number of language constructs that do not involve assignment have declarative semantics.

The following *Translation Process Table* associates with each such a language construct a corresponding formula of Definition 2. We denote by $\mathcal{T}(S)$ the translation of the sequence of statements S, where B denotes boolean values (`TRUE` or `FALSE`), and A an atomic statement, such as an equality or a relation symbol (with arguments):

Alma-0 Construct	Formula
B	B
A	A (atom)
`NOT` S	$\neg\mathcal{T}(S)$
S_1; S_2	$\mathcal{T}(S_1) \wedge \mathcal{T}(S_2)$
`IF` T `THEN` S `END`	$\mathcal{T}(T) \rightarrow \mathcal{T}(S)$
`IF` T `THEN` S_1 `ELSE` S_2 `END`	$(\mathcal{T}(T) \wedge \mathcal{T}(S_1)) \vee (\neg\mathcal{T}(T) \wedge \mathcal{T}(S_2))$
`EITHER` S_1 `ORELSE` S_2 `END`	$\mathcal{T}(S_1) \vee \mathcal{T}(S_2)$

`PROCEDURE p(MIX` $\bar{y} : \overline{\mathtt{T}}_1$`);`
`VAR` $\bar{x} : \overline{\mathtt{T}}_2$`;`
`BEGIN`
S
`END;`
(where $\overline{\mathtt{T}}_1$ and $\overline{\mathtt{T}}_2$ are the types of variables in $\bar{y}$ and $\bar{x}$, respectively.)
	$p(\bar{y}) \leftrightarrow \exists\, \bar{x}\ \mathcal{T}(S)$

`SOME` $i := s$ `TO` t `DO` S `END`	$\exists\, i \in [s..t]\ \mathcal{T}(S)$
`FOR` $i := s$ `TO` t `DO` S `END`	$\forall\, i \in [s..t]\ \mathcal{T}(S)$

We call the reader's attention here, that the $\wp$-axiom obtained by the translation of a procedure definition is considered universally closed with respect to the arguments of the left hand side ($\bar{y}$).

In the following, this translation allows us to work with specific formulas that represent Alma-0 programs.

There are several interesting remarks made explicit in [2] regarding this translation, from which we are recalling here only one that the reader may find relevant. Namely, that due to the use of generalised equality the sequences of statements `x = 0; y = x` and `y = x; x = 0;` are not equivalent. Consequently, the conjunction $\wedge$ is not commutative either, property being accomplished by the semantics of our formulas, as the computation tree of the formula $x = 0 \wedge y = x$ for the empty valuation yields a *success* leaf, whereas the computation tree of the formula $y = x \wedge x = 0$ for the empty valuation yields an *error* leaf.

Chapter 4

Tail-recursion

4.1 About Recursion

Recursion is a fundamental concept in mathematics and computer science and many practical computations can be fitted to a recursive framework. Simply, a recursive function (program or algorithm) is one which calls itself as part of the function body. An essential ingredient of recursion is that there must be a "termination condition", i.e. the call to oneself must be conditional to some test or predicate condition which will cease the recursive calling. A recursive program must cease the recursion on some condition or be in a circular state, i.e. an endless loop.

In a computer implementation of a recursive algorithm, a function will conditionally call itself. When any function call is performed, a new complete copy of the function's "information", such as parameters, return addresses, etc., is placed into general data and/or stack memory. When a function returns or exits, this information is returned to the free memory pool and the function ceases to actively exist. In recursive algorithms, many levels of function calls can be initiated resulting in many copies of the function being currently active and copies of the different functions' information residing in the memory spaces. Thus recursion provides no savings in storage nor will it be faster than a good non-recursive implementation. However, recursive code will often be more compact, easier to design, develop, implement, integrate, test, and debug.

The canonical example of a recursive function is factorial on $\mathbb{N}$, the following C-code represents an implementation for it:

```
int factorial (int n) {
  if (n == 0) return 1;
  else return n * factorial(n-1);
}
```

Functional programming languages rely heavily on recursion, using it where a procedural language would use iteration.

According to [12], when the last statement executed in the body of a procedure is a recursive call, this call is said to be *tail-recursive*. A procedure may make several recursive calls but a call is only tail-recursive if the caller returns immediately after it. A procedure as a whole is *tail-recursive* if all its recursive calls are tail-recursive.

We mention here that the function factorial from above is not tail-recursive. The following code presents a tail-recursive factorial function:

```
int factorial (int n) {
  return fact(n,1);
}
int fact (int n, int product) {
  if (n == 0) return product;
  else return fact(n-1,n * product);
}
```

The following code for determining the greatest common divisor of integers u and v is also tail-recursive (taken from [8]).

```
int gcd (int u, int v) {
  if (v == 0) return u;
  else return gcd(v,u%v);
}
```

4.2 Tail-recursion Optimisation

In case of a tail-recursive procedure or function, it is not necessary to retain the calling environment. This is important when a procedure calls itself recursively many times for, as without tail-recursion optimisation, the environments of earlier invocations would fill up the memory only to be discarded when (if) the last call terminated.

Normally when a procedure A calls procedures B,C, ..., Z, the environment of procedure A is only discarded when procedure Z returns and procedure A itself terminates. Using *last call optimisation*, A's environment is discarded as Z is called. This allows arbitrarily deep nesting of procedure calls without consuming memory to store useless environments.

Tail-recursion optimisation is a special case of last call optimisation but it allows the further optimisation that some arguments may be passed in situ, possibly in registers. It allows recursive functions to be compiled into iterative loops.

In general, when code is generated for a program, one operation that can sometimes be relatively expensive is procedure call, where many calling sequence operations must be performed. Modern processors have reduced this cost substantially by offering hardware support for standard calling sequences,

but the removal of frequent calls to small procedures can still produce measurable speedups. There are two standard ways to remove procedure calls. One is to replace the procedure call with the code for the procedure body (with suitable replacement of parameters by arguments). This is called *procedure inlining*, and sometimes is even a language option (as in C++). Another possible way to eliminate a procedure call is to recognize tail-recursion.

Tail-recursion is equivalent to assigning the values of the new arguments to the parameters and performing a jump to the beginning of the body of the procedure. For example the tail-recursive factorial function can be rewritten by the compiler to the equivalent code:

```
int fact (int n, int product) {
begin:
  if (n == 0) return product;
  else {
     n = n-1; product = n*product
     goto begin;
  }
}
```

This process is called *tail-recursion removal*. This example illustrates also very well that the memory usage can be optimised when a procedure is tail-recursive.

We can assume without any loss of generality that a compiler theoretically can be "smart" enough to recognize tail-recursion, and to proceed with tail-recursion optimisation, the practical question of "how to do it" remaining a pure technical matter rooted in the techniques of compiler construction.[8]

4.3 Alma-0 and Tail-recursion

The case studies of the following chapters present declarative solutions for well-known computational problems. All these solutions are written in the Alma-0 programming language that supports declarative programming. As the reader will see, in many cases when arguing about the possibility of reduced space-complexity we argue under the assumption of tail-recursion optimisation.

Unfortunately, the Alma-0 compiler [9] doesn't implement tail-recursion optimization.

In practice this optimisation technique for declarative programming (and in particular for the Alma-0 compiler) hasn't been implemented, but this remains only a technical matter to sort out, whereas its absence doesn't affect the overall theoretical results obtained in the following chapters.

Chapter 5

Maximum Element

Our first case study is concerned with the classical problem of finding a maximum element in a set of objects, A, with respect to a given total order: $\leq \subseteq A \times A$, defined over this set.

In our case we assume that A consists of a finite set of objects, $|A| = N$, with $N > 0$, and thus without any loss of generality we assume further on that we are able to refer to the these objects via indexing: $A = \{a_1, a_2, ..., a_N\}$, this having the only purpose to ease the reference to these objects, but not stating anything about the ordering between the objects.

The *maximum problem* for this set of elements is specified as follows: we are looking for an index $m \in [1..N]$ s.t. a_m is a maximum element w.r.t. the ordering, that is:

$$\mathbf{S} =_{def} \forall\, j \in [1..N]\ \ a_j \leq a_m \tag{5.1}$$

In the following we present three ALMA implementations that solve this problem in a purely declarative manner. The objects in this case are elements of an array of integer type, the total order is the one "less than or equal" relation on integer numbers. In each of the cases it is also provided a formal proof for the implication: $\mathbf{I} \rightarrow \mathbf{S}$, where $\mathbf{I}$ stands for the dual reading of the implementation as a formula.

5.1 Naive Solution

As a first approach we take advantage of the built-in backtracking mechanism of ALMA. The following program segment takes each element and compares it to all the other elements of the array. Upon encountering the first index that satisfies the specification, the program succeeds resulting the desired index in variable m.

```
TYPE Vect = ARRAY [1..N] OF INTEGER;
VAR i, j, m :  INTEGER;
    a :  Vect;
```

```
BEGIN
 SOME i := 1 TO N DO
   FOR j := 1 TO N DO
      a[j] <= a[i]
   END;
   m=i;
 END;
END;
```

The dual reading of the program defines the formula:

$$\exists\, i \in [1..N]\ (\forall\, j \in [1..N]\ a_j \leq a_i \wedge m = i)$$

which immediately implies the specification (5.1):

$$\forall\, j \in [1..N]\ \ a_j \leq a_m$$

Unfortunately, the worst-case running time of this approach is $O(N^2)$, whereas the space complexity is $O(N)$, the same as in the case of the imperative solution.

5.2 Reduced Time Complexity

We worked out an other declarative solution that uses an auxiliary array for storing some temporary information, namely the index of a "candidate" maximum element, thus increasing the space complexity to some extent, but that still being linear: $O(N)$, the worst-case running time in this case is $O(N)$. The required index is provided in variable m.

```
TYPE Vect = ARRAY [1..N] OF INTEGER;
     Index = ARRAY [1..N] OF [1..N];

VAR i,m :  INTEGER;
    a :  Vect;
    b :  Index;

BEGIN
 b[1] = 1;
 FOR i := 2 TO N DO
  IF a[b[i-1]] <= a[i] THEN b[i] = i
                       ELSE b[i] = b[i-1]
  END
 END;
 b[N] = m;
END;
```

The formula defined by this program is as follows:

$$\begin{aligned}\mathbf{I} =_{def}\ & b_1 = 1 \\ & \wedge\ \forall\, i \in [2..N]\ [(a_{b_{i-1}} \leq a_i\ \wedge\ b_i = i)\ \vee\ (a_{b_{i-1}} > a_i\ \wedge\ b_i = b_{i-1})] \\ & \wedge\ b_N = m\end{aligned} \tag{5.2}$$

We have to prove that:

$$\mathbf{I} \rightarrow\ \forall\, j \in [1..N]\ a_j \leq a_m$$

First we state and prove the following Lemma:

Lemma 5.2.1 With **I** defined as in (5.2) we have:

$$\mathbf{I} \rightarrow\ \forall\, i \in [1..N]\ \forall\, j \in [1..i]\ a_j \leq a_{b_i}$$

Proof: We follow mathematical induction on the range of i:
Base case: $i = 1$

$$\begin{aligned}\mathbf{I}\ \ \rightarrow\ \ & (b_1 = 1) \rightarrow a_1 = a_{b_1} \rightarrow a_1 \leq a_{b_1} \rightarrow \forall\, j \in [1..1]\ a_j \leq a_{b_1} \\ & \rightarrow \forall\, i \in [1..1]\ \forall\, j \in [1..i]\ a_j \leq a_{b_i}\end{aligned}$$

Thus the base case holds.
Induction step:
Hypothesis: We presume it is true for any $i \leq k$, $k < N$:

$$\mathbf{I} \rightarrow\ \forall\, i \in [1..k]\ \forall\, j \in [1..i]\ a_j \leq a_{b_i} \tag{5.3}$$

and we prove for $i = k + 1$, that:

$$\mathbf{I} \rightarrow\ \forall\, j \in [1..k+1]\ a_j \leq a_{b_{k+1}} \tag{5.4}$$

From **I** with $i = k + 1$ we have:

$$\mathbf{I} \rightarrow (a_{b_k} \leq a_{k+1}\ \wedge\ b_{k+1} = k+1)\ \vee\ (a_{b_k} > a_{k+1}\ \wedge\ b_{k+1} = b_k)$$

We distinguish two cases according to the disjunction from above:
(a) From (5.3) we have: $\forall\, j \in [1..k]\ a_j \leq a_{b_k}$ then by the first part of the disjunction:

$$\forall\, j \in [1..k]\ a_j \leq a_{k+1} \leq a_{b_{k+1}} \tag{5.5}$$

for $j = k + 1$

$$a_{k+1} \leq a_{k+1} = a_{b_{k+1}} \tag{5.6}$$

By (5.5) and (5.6) we conclude (5.4).

(b) From (5.3) and the second part of the disjunction we have:

$$\forall\, j \in [1..k]\ a_j \leq a_{b_k} = a_{b_{k+1}} \tag{5.7}$$

for $j = k + 1$:

$$a_{k+1} < a_{b_k} = a_{b_{k+1}} \tag{5.8}$$

By (5.7) and (5.8) we conclude (5.4).

As both cases imply (5.4) the induction step holds, thus the property holds for any $i \leq N$. □

Proposition 5.2.2. With **I** defined as in (5.2) we have:

$$\mathbf{I} \rightarrow \forall\, j \in [1..N]\ a_j \leq a_m$$

Proof: Directly from Lemma 5.2.1. with $i = N$, and from **I** we have $b_N = m$. □

Thus we have our proof for $\mathbf{I} \rightarrow \mathbf{S}$ complete.

5.3 Tail-recursive Approach

In this case we are using a tail-recursive procedure. Considering the possibility of optimized memory usage when tail-recursive procedures are present, this solution will have a reduced space complexity compared to the previous solution, in the sense that it is still $O(N)$, but the asymptotic constant here is much closer to the one of the imperative solution, whereas the time complexity is still $O(N)$.

Here instead of storing the indices of all the elements that were once "candidate" maximum elements, we pass the current one on to the next procedure call, this latter in the same time getting the value of i increased, the parameter by which we parse the whole array during the successive calls of the procedure.

```
TYPE Vect = ARRAY [1..N] OF INTEGER;
VAR m :  INTEGER;
    a :  Vect;

PROCEDURE Maxim(b :  Vect; i,j :  INTEGER; VAR p :  INTEGER);
BEGIN
  IF i = N+1 THEN p = j
             ELSE IF b[i] > b[j] THEN Maxim(b,i+1,i,p)
                                 ELSE Maxim(b,i+1,j,p)
                  END;
  END
END Maxim;
```

The problem is solved by calling:

```
Maxim(a,1,1,m);
```

the requested index being provided in variable m, which has been accumulated during the recursive calls.

We call the reader's attention that this is a terminating recursive procedure. It is easy to verify that when the procedure is called for a parameter $i \in [1..N]$ recursion takes place until the actual parameter i hits the value $N+1$. In all the other cases a runtime error will arise.

The following formula constitutes the declarative interpretation of the definition of procedure `Maxim`:

$$M(b,i,j,p) \leftrightarrow (i = N+1 \ \wedge \ p = j) \ \vee \ (i \neq N+1 \wedge \\ \wedge \ ((b_i > b_j \ \wedge \ M(b,i+1,i,p)) \vee (b_i \leq b_j \ \wedge \ M(b,i+1,j,p))) \tag{5.9}$$

Then the declarative interpretation of the whole implementation is defined as follows:

$$\mathbf{I} =_{def} M(b,i,j,p) \leftrightarrow (i = N+1 \ \wedge \ p = j) \ \vee \ (i \neq N+1 \wedge \\ \wedge \ ((b_i > b_j \ \wedge \ M(b,i+1,i,p)) \vee (b_i \leq b_j \ \wedge \ M(b,i+1,j,p)) \tag{5.10}$$

$$\wedge$$

$$M(a,1,1,m) \tag{5.11}$$

Proposition 5.3.1. With **I** defined as in (5.10) we have:

$$\mathbf{I} \rightarrow \forall\, i \in [1..N] \ \ a_i \leq a_m$$

Proof: We follow induction on N:
Base case: $N = 1$

$$\begin{aligned} \mathbf{I} \ \rightarrow \ & (M(a,1,1,m) \rightarrow M(a,2,1,m)) \\ & M(a,2,1,m) \leftrightarrow (2 = 2 \ \wedge \ m = 1) \ \vee \ (2 \neq 2) \ \wedge \ [...] \Rightarrow m = 1 \end{aligned}$$

We conclude: $\forall\, i \in [1..1] \ \ a_i \leq a_m$, that is $a_1 \leq a_1$, which is true. The base case holds.
Induction step:
Hypothesis: Presume that for $N = k$ $(k > 0)$ we have:

$$\mathbf{I} \rightarrow \forall\, i \in [1..k] \ \ a_i \leq a_m$$

We prove for $N = k+1$:

$$\mathbf{I} \rightarrow \forall\, i \in [1..k+1] \ \ a_i \leq a_m$$

By definition (5.10), applying the formula $M(a,1,1,m)$ will result in applying a series of $M(a,i,j,m)$'s with variables i,j,m accumulated and/or updated during the recursion. This series up to $i = k$ will be the very same as in case when N would equal k, and thus the resulting m would get the value of j, where $M(a,k+1,j,m)$ is the last application of M.

Then by (5.9): $M(a,k+1,j,m) \rightarrow m = j$, which by hypothesis leads to:

$$\forall\, i \in [1..k] \ \ a_i \leq a_j \tag{5.12}$$

As now we have $n = k+1$, $M(a,k+1,j,m)$ will yield a new application of M, that is $M(a,k+2,l,m)$, where

$$l = j, \text{ if } a_{k+1} \leq a_j \text{ or}$$

$$l = k + 1, \text{ if } a_{k+1} > a_j$$

As this will be the last application, we get $m = l$, and so by (5.12) we conclude that the induction step holds.

Thus for any $N \geq 1$ we have:

$$\mathbf{I} \to \forall\, i \in [1..N]\ \ a_i \leq a_m.$$

Chapter 6

Bubblesort

In this chapter we have elaborated purely declarative solutions for the popular sorting algorithm "Bubblesort".

Just as in the previous section we are working with elements of a finite set endowed with a total ordering, being able to refer to these elements via indexing. The following implementations work on an array of integer type, and the total order is the one "less than or equal" relation on integer numbers.

Under these assumptions we specify the *sorting problem* as follows:

Given the *input data*, array $a[i]$, $i \in [1..N]$, we are looking for the *output data*, array $b[i]$, $i \in [1..N]$, such that b is a sorted permutation of a.

$$\begin{aligned} \mathbf{S}(a,b) \quad =_{def} \quad & 'b \text{ is a sorted permutation of } a' \Leftrightarrow \\ & \forall i \in [1..N-1] : \; b[i] \leq b[i+1] \; \wedge \\ & \exists \text{ bijective mapping } \pi : [1..N] \rightarrow [1..N] : \\ & b[i] = a[\pi[i]], \; \forall i \in [1..N] \end{aligned} \tag{6.1}$$

Bubblesort works by repeatedly swapping adjacent elements that are out of order. The classical imperative solution can be expressed by the following pseudo-code:

for $i = 1$ **to** $N - 1$ **do**
  for $j = 1$ **to** $N - i$ **do**
    if $a[j] > a[j+1]$ **then** $swap(a[j], a[j+1])$

The algorithm has a time complexity of $O(N^2)$, and the space required is $O(N)$.

If we analyse a little bit this imperative pseudo-code, we can easily convince ourselves, that the procedure `swap(x,y)`as its name suggests, will contain destructive assignments. Just simply by the fact, that it will make x have the value of y, and y have the value of x. Such assignments are not allowed in declarative programming, as there we abstract from a computational notion like state, therefore during the whole computation a variable will not change

its value. This is indeed essential, as otherwise the formula-reading of the program would have truth value False. To make it more explicit, let's consider the following program segment:

```
x := 3;
x := 5;
```

Its translation into a formula would be: $x = 3 \wedge x = 5$, which is obviously False, whereas the program segment simply assigns first 3 to x, and then 5.

On the other hand, it is also worth pointing out, that in the imperative version this swapping procedure is essential, as the original Bubblesort is an in-situ sorting algorithm, which cannot be the case in declarative style, as we have seen.

So, in our declarative implementations we cannot use such a procedure, nor can we sort in place, but we can always require more space in order to be able to avoid destructive assignments, which, unfortunately, may increase space complexity.

The following implementations also give some hints on how to achieve optimal space complexity and yet insisting on declarative semantics.

6.1 Using Matrices

In the above pseudo-code after each iteration of the outer loop at the $(i + 1)$th position there will be placed the maximum of the elements from before and including the $(i + 1)$th element itself, this being achieved by repeatedly swapping adjacent elements that are out of order. In our solution instead of swapping, we require a new array at each iteration of the outer loop in order to "simulate" Bubblesort, and also an array to be able to store maximum values, as follows: at each iteration of the inner loop we will store in this array the maximum of the elements already visited in that line. This will be compared to the next element, obtaining this way the new maximum element to be placed to the next position in this auxiliary array. Once we have finished with the $(N - i)$th iteration of the inner loop, we will be able to place the maximum element of the $N - i + 1$ elements in its proper place in the array used for simulating.

So altogether, we require an $N \times N$-sized matrix a to imitate Bubblesort, and an auxiliary $N \times N$-sized matrix b to store the maximum elements as discussed above. The initial array to be sorted is given in $a[1]$, whereas in $a[N]$ we will get the sorted permutation of $a[1]$.

```
TYPE Vect = ARRAY [1..N] OF INTEGER;
     Mat = ARRAY [1..N] OF Vect;

VAR  i, j :  INTEGER;
```

```
      a,b :  Mat;

PROCEDURE Min_Max(x,y :  INTEGER; VAR min,max :  INTEGER);
BEGIN
  IF x <= y THEN min = x;
                 max = y
            ELSE min = y;
                 max = x
  END;
END Min_Max;

BEGIN
  FOR i := 1 TO N-1 DO
     a[i,1] = b[i,1];
     FOR j := 2 TO N-i+1 DO
        Min_Max(b[i,j-1],a[i,j],a[i+1,j-1],b[i,j]);
     END;
     a[i+1,N-i+1] = b[i,N-i+1];
   END;

   FOR i := 3 TO N DO
     a[N,i] = a[N-i+2,i];
   END;
END.
```

Note, that we do not waste time for copying into the next line of the matrix the maximum elements already placed at their proper place, instead we leave this operation as a last step, when the final sorted array $a[N]$ is completed.

The following matrix shows how the implementation works on $a[1]$, an array of 12 elements:

$a[1]$:	10	2	4	-5	-11	-1	9	-20	105	0	75	-8
$a[2]$:	2	4	-5	-11	-1	9	-20	10	0	75	-8	105
$a[3]$:	2	-5	-11	-1	4	-20	9	0	10	-8	75	
$a[4]$:	-5	-11	-1	2	-20	4	0	9	-8	10		
$a[5]$:	-11	-5	-1	-20	2	0	4	-8	9			
$a[6]$:	-11	-5	-20	-1	0	2	-8	4				
$a[7]$:	-11	-20	-5	-1	0	-8	2					
$a[8]$:	-20	-11	-5	-1	-8	0						
$a[9]$:	-20	-11	-5	-8	-1							
$a[10]$:	-20	-11	-8	-5								
$a[11]$:	-20	-11	-8									
$a[12]$:	-20	-11	-8	-5	-1	0	2	4	9	10	75	105

and the auxiliary matrix b corresponding to a:

$b[1]$:	10	10	10	10	10	10	10	10	105	105	105	105
$b[2]$:	2	4	4	4	4	9	9	10	10	75	75	
$b[3]$:	2	2	2	2	4	4	9	9	10	10		
$b[4]$:	-5	-5	-1	2	2	4	4	9	9			
$b[5]$:	-11	-5	-1	-1	2	2	4	4				
$b[6]$:	-11	-5	-5	-1	0	2	2					
$b[7]$:	-11	-11	-5	-1	0	0						
$b[8]$:	-20	-11	-5	-1	-1							
$b[9]$:	-20	-11	-5	-5								
$b[10]$:	-20	-11	-8									
$b[11]$:	-20	-11										

Space complexity is therefore $O(N^2)$, as well as time complexity, this latter being the same as in the imperative solution.

Note: in case we have to deal with big-sized objects for which it would be costly copying array elements, the whole algorithm can be easily rewritten to manipulate on indices instead of whole elements. In that case, we would require matrices a and b to be index matrices.

This implementation contains no destructive assignments and has a pure declarative semantics, its declarative interpretation being the formula:

$$
\begin{aligned}
\mathbf{I} =_{def} & M(x, y, min, max) \leftrightarrow (x \leq y \wedge min = x \wedge max = y) \vee \\
& (x > y \wedge min = y \wedge max = x) && (6.2) \\
\wedge & \\
& \forall i \in [1..N-1] \\
& [a[i,1] = b[i,1] \wedge && (6.3) \\
& \forall j \in [2..N-i+1] \\
& M(b[i,j-1], a[i,j], a[i+1,j-1], b[i,j]) \wedge && (6.4) \\
& a[i+1, N-i+1] = b[i, N-i+1]] && (6.5) \\
\wedge & \\
& \forall i \in [3..N] \\
& a[N,i] = a[N-i+2, i] && (6.6)
\end{aligned}
$$

We have to prove the implication:

$$\forall a : \text{Mat } (\mathbf{I} \rightarrow \mathbf{S}(a[1], a[N]))$$

First, we need to prove several lemmas.

Lemma 6.1.1. Assuming $\mathbf{I}$, this implies the following:

$$
\begin{aligned}
\mathbf{I} \rightarrow \quad & \forall i \in [1..N-1] \; \forall j \in [1..N-i+1] \\
& b[i,j] = max\,\{a[i,1], a[i,2], ..., a[i,j]\}
\end{aligned}
$$

Proof:
For an arbitrary but fixed i we follow induction on j:
Base case: $j = 1$. From (6.3) it is straightforward.
Induction step:
Hypothesis: Presume for $j = k < N - i + 1$:

$$b[i,k] = max\,\{a[i,1], a[i,2], ..., a[i,k]\}$$

We prove it for $j = k + 1$:
From (6.2) and (6.4) we conclude:

$$\begin{aligned} b[i,k+1] &= max\,\{b[i,k], a[i,k+1]\} \\ &=_H max\,\{a[i,1], a[i,2], ..., a[i,k+1]\} \end{aligned}$$

which yields that the induction step holds.
Thus the property holds for any $j \in [1..N - i + 1]$ and arbitrary $i \in [1..N - 1]$. □

Corollary 6.1.2. Assuming **I**, this implies the following

$$\begin{aligned} \mathbf{I} \rightarrow \quad & \forall i \in [1..N-1] \\ & a[i+1, N-i+1] = max\,\{a[i,1], a[i,2], ..., a[i,N-i+1]\} \end{aligned}$$

Proof:
Directly from Lemma 6.1.1. with $j = N - i + 1$ and (6.5). □

Lemma 6.1.3. Assuming **I**, this implies the following:

$$\begin{aligned} \mathbf{I} \rightarrow \quad & \forall i \in [1..N-1]\ \forall j \in [1..N-i+1] \\ & a[i+1,1], ..., a[i+1,j-1], b[i,j] \text{ is a permutation of} \\ & a[i,1], ..., a[i,j-1], a[i,j] \end{aligned}$$

Proof:
For an arbitrary but fixed i we follow induction on j:
Base case: $j = 1$. From (6.3), it's trivial.
Induction step:
Hypothesis: Presume it is true for $j = k < N - i + 1$, that is:

$$\begin{aligned} & a[i+1,1], ..., a[i+1,k-1], b[i,k] \text{ is a permutation of} \\ & a[i,1], ..., a[i,k-1], a[i,k] \end{aligned}$$

We prove for $j = k + 1$ that:

$$\begin{aligned} & a[i+1,1], ..., a[i+1,k-1], a[i+1,k], b[i,k+1] \text{ is a permutation of} \\ & a[i,1], ..., a[i,k-1], a[i,k], a[i,k+1] \end{aligned}$$

From (6.2) and (6.4) we have that the set of elements$\{a[i+1,k], b[i,k+1]\}$is the same set as $\{b[i,k], a[i,k+1]\}$. This and the assumption of the hypothesis yields directly the induction step.

Thus the property holds for any $j \in [1..N-i+1]$ and arbitrary $i \in [1..N-1]$. □

Corollary 6.1.4. Assuming **I**, this implies the following:

$$\begin{aligned}\mathbf{I} \rightarrow \quad & \forall i \in [1..N-1] \\ & a[i+1,1], ..., a[i+1,N-i], a[i+1,N-i+1] \\ & \text{is a permutation of} \\ & a[i,1], ..., a[i,N-i], a[i,N-i+1]\end{aligned}$$

Proof:
Straightforward from (6.5) and Lemma 6.1.3. with $j = N-i+1$. □

Corollary 6.1.5. Assuming **I**, this implies the following, $\forall i \in [1..N-1]$:
a)

$$a[i+1,1], ..., a[i+1,N-i+1], a[i,N-i+2], ..., a[2,N]$$
$$\text{is a permutation of } a[1]$$

b)

$$a[i+1,N-i] \leq a[i+1,N-i+1] \leq a[i,N-i+2], ..., a[2,N]$$

Proof:
a) Induction on i**:**
Base case: $i = 1$. Directly from Cor. 6.1.4. with $i = 1$.
Induction step:
Hypothesis: Assume, the property is true for $i = k < N-1$:

$$a[k+1,1], ..., a[k+1,N-k+1], a[k,N-k+2], ..., a[2,N]$$
$$\text{is a permutation of } a[1]$$

We have to prove for $i = k+1$ that:

$$a[k+2,1], ..., a[k+2,N-k], a[k+1,N-k+1], a[k,N-k+2], ..., a[2,N]$$
$$\text{is a permutation of } a[1]$$

From Cor. 6.1.4. with $i = k+1$ we have that:

$$\begin{aligned}& a[k+2,1], ..., a[k+2,N-k] \text{ is a permutation of} \\ & a[k+1,1], ..., a[k+1,N-k]\end{aligned}$$

Adding the elements $a[k+1, N-k+1], a[k, N-k+2], ..., a[2, N]$ to both series, under the assumption of the hypothesis we have that the induction step holds.

The property holds for any $i \in [1..N-1]$.

b) We follow again induction on i:

Base case: $i = 1$. From Cor. 6.1.2. and 6.1.4. with $i = 1$ we have

$$\begin{aligned} a[2, N] &= max\{a[1,1], a[1,2], ..., a[1,N]\} = \\ &= max\{a[2,1], a[2,2], ..., a[2,N]\} \end{aligned}$$

From this we conclude $a[2, N-1] \leq a[2, N]$, thus the base case holds.

Induction step:

Hypothesis: We presume it is true for $i = k < N-1$:

$$a[k+1, N-k] \leq a[k+1, N-k+1] \leq a[k, N-k+2], ..., a[2, N]$$

We have to prove it for $i = k+1$:

$$a[k+2, N-k-1] \leq a[k+2, N-k] \leq a[k+1, N-k+1], ..., a[2, N]$$

From Cor. 6.1.2. with $i = k+1$ and Cor. 6.1.4. with $i = k+1$ we have:

$$\begin{aligned} a[k+2, N-k] &= max\{a[k+1,1], ..., a[k+1, N-k]\} = \\ &= max\{a[k+2,1], ..., a[k+2, N-k]\} \end{aligned}$$

$$\rightarrow a[k+2, N-k-1] \leq a[k+2, N-k] \tag{6.7}$$

From Cor. 6.1.2. with $i = k$ and Cor. 6.1.4. with $i = k$ and later $i = k+1$ we have:

$$\begin{aligned} a[k+1, N-k+1] &= max\{a[k,1], ..., a[k, N-k+1]\} = \\ &= max\{a[k+1,1], ..., a[k+1, N-k+1]\} \end{aligned}$$

$$\begin{aligned} \rightarrow a[k+1, N-k+1] \geq\ & max\{a[k+1,1], ..., a[k+1, N-k]\} \\ =\ & max\{a[k+2,1], ..., a[k+2, N-k]\} \end{aligned}$$

$$\rightarrow a[k+2, N-k] \leq a[k+1, N-k+1] \tag{6.8}$$

(6.7), (6.8), and the hypothesis yield the induction step.

Thus the property is true for any $i \in [1..N-1]$. □

Theorem 6.1.6. Assuming **I**, this implies that $a[N]$ is a sorted permutation of $a[1]$.

Proof:

From Cor.6.1.5. with $i = N-1$ we have:

$$\begin{gathered} a[N,1], a[N,2], a[N-1,3], ..., a[2,N] \\ \text{is a sorted permutation of } a[1] \end{gathered}$$

This and (6.6) yield that $a[N]$ is a sorted permutation of $a[1]$. □

Thus we have our proof for $\mathbf{I} \rightarrow \mathbf{S}(a[1], a[N])$ complete.

6.2 Reducing Space

As Lemma 6.1.1. clearly stated, in each line of the auxiliary matrix, b, there are stored maximum values of elements up to a certain index from the same line of the main matrix a. Using a whole matrix to store these values has proved inefficient. We have seen that each line of this main matrix is used to "simulate" Bubblesort, so apparently we would need just one auxiliary array of dimension N, for each new line to be computed of the main matrix, and for this job we may reuse this array. In this present proposal we hide the computation of a new line in a procedure (`Calculate_New_Line`), requesting here a local variable of dimension N, array b.

Though this way we may reduce to some extent the space needed, however its magnitude is still quadratic in N (because of the matrix a), and so is the time complexity.

This implementation uses procedure `Min_Max`, as well, defined as in the previous section.

```
TYPE Vect = ARRAY [1..N] OF INTEGER;
Mat = ARRAY [1..N] OF Vect;
VAR  i, j :  INTEGER; a :  Mat;

[PROCEDURE Min_Max...]

PROCEDURE Calculate_New_Line (c:  Mat; i:  INTEGER);
VAR b:  Vect;
BEGIN
  c[i,1] = b[1];
  FOR j := 2 TO N-i+1 DO
     Min_Max(b[j-1],c[i,j],c[i+1,j-1],b[j]);
  END;
  c[i+1,N-i+1] = b[N-i+1];
END Calculate_New_Line;

BEGIN
  FOR i := 1 TO N-1 DO
    Calculate_New_Line(a,i);
  END;
  FOR i := 3 TO N DO
    a[N,i] = a[N-i+2,i];
  END;
END.
```

This implementation gets translated into the following formula:

$$\begin{aligned}
\mathbf{I} =_{def} & M(x,y,min,max) \leftrightarrow (x \leq y \wedge min = x \wedge max = y) \vee \qquad (6.9)\\
& (x > y \wedge min = y \wedge max = x)\\
\wedge\\
& C(c,i) \leftrightarrow \exists\, b\, [(c[i,1] = b[1]) \wedge \qquad (6.10)\\
& \quad \forall j \in [2..N-i+1]\\
& \quad\quad M(b[j-1], c[i,j], c[i+1,j-1], b[j]) \wedge \qquad (6.11)\\
& \quad c[i+1, N-i+1] = b[N-i+1])] \qquad (6.12)\\
\wedge\\
& \forall i \in [1..N-1]\\
& \quad C(a,i) \qquad (6.13)\\
\wedge\\
& \forall i \in [3..N]\\
& \quad a[N,i] = a[N-i+2,i] \qquad (6.14)
\end{aligned}$$

Again, in terms of formulae we have to prove the following:

$$\forall a : \mathrm{Mat}\ (\mathbf{I} \rightarrow \mathbf{S}(a[1], a[N]))$$

where S is the specification for sorting defined in (6.1).

Lemma 6.2.1. Assuming $\mathbf{I}$, this implies the following:

$$\begin{aligned}
\mathbf{I} \rightarrow \quad & \forall i \in [1..N-1]\ \exists b : Vect\\
& \forall j \in [1..N-i+1]\\
& \quad b[j] = max\,\{a[i,1], a[i,2], ..., a[i,j]\}\\
& \wedge\\
& a[i+1, N-i+1] = b[N-i+1]
\end{aligned}$$

Proof: For an arbitrary but fixed i applying the definition of $C(a,i)$(6.10) we may follow mathematical induction on j similar to that of the proof of Lemma 6.1.1. We call the reader's attention of the semantical analogy between (6.3) and (6.10), (6.4) and (6.11), (6.5) and (6.12). □

This directly leads us to the result of Cor. 6.1.2. on page 56. Note that this result doesn't depend anymore on the existentially quantified variable b.

Likewise, we can formulate the counterpart of Lemma 6.1.3. which leads again directly to Cor. 6.1.4.

Lemma 6.2.2. Assuming **I**, this implies the following:

$$
\begin{aligned}
\mathbf{I} \Rightarrow \quad & \forall i \in [1..N-1]\ \exists b : Vect \\
& \quad \forall j \in [1..N-i+1] \\
& \quad\quad a[i+1,1], ..., a[i+1,j-1], b[j] \text{ is a permutation of} \\
& \quad\quad a[i,1], ..., a[i,j-1], a[i,j] \\
& \quad \wedge \\
& \quad a[i+1, N-i+1] = b[N-i+1]
\end{aligned}
$$

Proof: For an arbitrary but fixed i applying the definition of $C(a,i)$(6.10) we may follow mathematical induction on j similar to that of the proof of Lemma 6.1.3. □

We can conclude therefore that the properties formulated in Cor. 6.1.2. and Cor. 6.1.4. hold in this case as well, both are being "free" of the existential quantifier.

Therefore from this point on, the proof for $\mathbf{I} \rightarrow \mathbf{S}$ is identical to the one provided in section 6.1., as the second part of the proof there provided (Cor. 6.1.5. and Theorem 6.1.6.) is based exclusively on the results of these two corollaries and (6.6) which happens to be the very same formula as (6.14).

6.3 Optimization by Tail Recursion

In the following we are organising the procedure `Calculate_New_Line` into a tail-recursive one. Under the assumption of optimized memory usage by tail-recursive procedures, we can achieve a space complexity linear in N, similar to the one of imperative solution.

The new procedure is called `Recursive_Calculate_New_Line` with parameters *i, a, a_sorted*. The main matrix used in the previous sections becomes "virtual" in the sense, that we will always focus just on one specific line of it at a time, and forget about the lines above. In the parameter list i indicates the number of this line, a is the actual array we work on, and *a_sorted* is an accumulated parameter, in which, when recursion finishes, the desired sorted permutation of the original array will be provided. The procedure has two local variables: array *a_next*, used to compute in it the next line of the "virtual" matrix for which the procedure will call itself, and an array b with similar task as before.

```
TYPE Vect = ARRAY [1..N] OF INTEGER;
VAR  j :  INTEGER; a,a_sorted :  Vect;

[PROCEDURE Min_Max...]
```

```
PROCEDURE Recursive_Calculate_New_Lines (i:INTEGER; c:  Vect;
                                                    VAR a_sorted:  Vect);
VAR
   a_next, b:  Vect;

BEGIN
  IF i <= N THEN
              c[1] = b[1];
              FOR j := 2 TO N-i+1 DO
                  Min_Max(b[j-1],c[j],a_next[j-1],b[j]);
              END;
              a_sorted[N-i+1] = b[N-i+1];
              Recursive_Calculate_New_Lines(i+1,a_next,a_sorted);
            END;
END Recursive_Calculate_New_Lines;

BEGIN
   Recursive_Calculate_New_Lines(1,a,a_sorted);
END;
```

The array to be sorted is in a, this is given as actual parameter in the first call of the procedure, 1 indicates that this is the first line in the virtual matrix. It is easy to verify that we have to deal here with a terminating recursive procedure: when it is called for a parameter $i \in [1..N]$ recursion takes place, and eventually when i hits the value $N+1$, recursion stops, as for values $i > N$ nothing happens. For values $i < 1$ runtime error will arise.

The implementation gets translated into the following formula:

$$
\begin{aligned}
\mathbf{I} =_{def}\ & M(x,y,min,max) \leftrightarrow (x \le y \wedge min = x \wedge max = y) \vee && (6.15)\\
& \quad (x > y \wedge min = y \wedge max = x) \\
& \wedge \\
& \quad R(i,c,a\_sorted) \leftrightarrow \exists\, a\_next, b\ [i \le N \rightarrow && (6.16)\\
& \qquad c[1] = b[1] \wedge && (6.17)\\
& \qquad \forall j \in [2..N-i+1] \\
& \qquad\quad M(b[j-1], c[j], a\_next[j-1], b[j]) \wedge && (6.18)\\
& \qquad a\_sorted[N-i+1] = b[N-i+1] \wedge && (6.19)\\
& \qquad R(i+1, a\_next, a\_sorted)] && (6.20)\\
& \wedge \\
& \quad R(1,a,a\_sorted)
\end{aligned}
$$

which leaves for us to prove the following implication:

$$\forall a, a\_sorted : \text{Vect}\ \ (\mathbf{I} \rightarrow \mathbf{S}(a, a\_sorted))$$

The following lemma is the counterpart of Lemma 6.2.1., its proof follows the same pattern, page 61.

Lemma 6.3.1. Assuming **I**, this implies the following:

$$
\begin{aligned}
R(i, c, a\_sorted) \rightarrow \quad & \exists b \ [i \leq N \ \rightarrow \\
& \quad \forall\, j \in [1..N - i + 1] : \\
& \qquad b[j] = max\,\{c[1], c[2], ..., c[j]\} \\
& \quad \wedge \\
& \quad a\_sorted[N - i + 1] = b[N - i + 1]]
\end{aligned}
$$

Corollary 6.3.2. Assuming **I**, this implies the following:

$$
\begin{aligned}
R(i, c, a\_sorted) \rightarrow \quad & [i \leq N \ \rightarrow \\
& a\_sorted[N - i + 1] = max\,\{c[1], c[2], ..., c[N - i + 1]\}]
\end{aligned}
$$

Proof: Directly from Lemma 6.3.1. □

The following lemma is the counterpart of Lemma 6.2.2., its proof follows the same pattern as there provided.

Lemma 6.3.3. Assuming **I**, this implies the following:

$$
\begin{aligned}
R(i, c, a\_sorted) \rightarrow \quad & \exists\, a\_next, b \\
& [i \leq N \ \rightarrow \\
& \quad \forall\, j \in [1..N - i + 1] : \\
& \qquad a\_next[1], ..., a\_next[j - 1], b[j] \text{ is a permutation of} \\
& \qquad c[1], ..., c[j - 1], c[j] \\
& \quad \wedge \\
& \quad a\_sorted[N - i + 1] = b[N - i + 1]]
\end{aligned}
$$

Corollary 6.3.4. Assuming **I**, this implies the following:

$$
\begin{aligned}
R(i, c, a\_sorted) \rightarrow \quad & \exists\, a\_next \\
& [i \leq N \ \rightarrow \\
& \quad a\_next[1], ..., a\_next[N - i], a\_sorted[N - i + 1] \\
& \quad \text{is a permutation of } c[1..N - i + 1]]
\end{aligned}
$$

Proof: Directly from Lemma 6.3.3. □

Theorem 6.3.5. Assuming **I**, this implies the following:[1]

$$
\begin{aligned}
R(1, a, a\_sorted) \rightarrow \quad & \forall\, i \leq N \;\; \exists\, a' \\
& \quad a\_sorted[N-i+1..N] \text{ is sorted} \\
& \wedge \\
& \quad a'[1..N-i] \text{ together with } a\_sorted[N-i+1..N] \\
& \quad \text{is a permutation of } a[1..N] \\
& \wedge \\
& \quad a\_sorted[N-i+1] \geq max\,\{a'[1..N-i]\} \\
& \wedge \\
& \quad R(i+1, a', a\_sorted)
\end{aligned}
$$

Proof: we follow induction on i.
Base case: $i = 1$, then first, $a\_sorted[N..N]$ is trivially sorted.
Second, by Cor.6.3.4. $\exists\, a\_next$ such that

$$
\begin{aligned}
& a\_next[1..N-1] \text{ together with } a\_sorted[N..N] \\
& \quad \text{is a permutation of } a[1..N].
\end{aligned}
$$

Third, by Cor.6.3.2. and Cor.6.3.4. with $i = 1$, $c = a$ we have:

$$a\_sorted[N] = max\,\{a[1..N]\} = max\,\{a\_next[1..N-1], a\_sorted[N]\}$$

Which results:

$$a\_sorted[N] \geq max\,\{a\_next[1..N-1]\}$$

Fourth, $R(2, a\_next, a\_sorted)$ follows directly from $R(1, a, a\_sorted)$ (6.16).
Thus, the base case holds.
Induction step:
Hypothesis: we presume that the implication is true for $i = k < N$:

$$
\begin{aligned}
R(1, a, a\_sorted) \rightarrow \quad & \exists\, a' \\
& \quad a\_sorted[N-k+1..N] \text{ is sorted} && (6.21) \\
& \wedge \\
& \quad a'[1..N-k] \text{ together with } a\_sorted[N-k+1..N] \\
& \quad \text{is a permutation of } a[1..N] && (6.22) \\
& \wedge \\
& \quad a\_sorted[N-k+1] \geq max\,\{a'[1..N-k]\} \\
& \wedge && (6.23) \\
& \quad R(k+1, a', a\_sorted) && (6.24)
\end{aligned}
$$

[1]Note: when the elements of an array, say b, are denoted by $b[x..y]$ with $x > y$, then $b[x..y]$ denotes no elements of b.

We have prove that it is true for $k+1$:

By hypothesis, $R(1, a, a\_sorted) \rightarrow R(k+1, a', a\_sorted)$, which implies that $\exists\, a\_next$ with several properties listed below, they follow directly from Cor.6.3.2. and Cor.6.3.4. with $R(k+1, a', a\_sorted)$:

First, by hypothesis we have:

$$a\_sorted[N-k+1] \geq max\,\{a'[1..N-k]\} = \\ max\,\{a\_next[1], ..., a\_next[N-k-1], a\_sorted[N-k]\}$$

This implies, that:

$$a\_sorted[N-k] \leq a\_sorted[N-k+1]$$

therefore by hypothesis (6.21) we conclude $a\_sorted[N-k..N]$ is sorted.

Second, as $a'[1..N-k]$ together with $a\_sorted[N-k+1..N]$ is a permutation of $a[1..N]$ by hypothesis (6.22), and that $a\_next[1], ..., a\_next[N-k-1], a\_sorted[N-k]$ is a permutation of $a'[1..N-k]$ (by Cor.6.3.4.), we can conclude that $a\_next[1..N-k-1]$ together with $a\_sorted[N-k..N]$ is a permutation of the original array, $a[1..N]$.

Third, by Cor.6.3.2. and Cor.6.3.4. with $R(k+1, a', a\_sorted)$ we have:

$$a\_sorted[N-k] = max\,\{a'[1..N-k]\} = \\ max\,\{a\_next[1..N-k-1], a\_sorted[N-k]\}$$

Which results:

$$a\_sorted[N-k] \geq max\,\{a\_next[1..N-k-1]\}\,.$$

Fourth, $R(k+2, a\_next, a\_sorted)$ follows directly from $R(k+1, a', a\_sorted)$ (6.16).

Thus, the induction step holds.

The implication is true for any $i \leq N$. □

Corollary 6.3.6. Assuming **I**, this implies the following:

$$R(1, a, a\_sorted) \rightarrow a\_sorted[1..N] \text{ is a sorted permutation of } a[1..N]$$

Proof: Directly by Theorem 6.3.5. for $i = N$, as $a'[1..0]$ is empty. □

Thus we have proved:

$$\forall a, a\_sorted : \text{Vect} \quad (\mathbf{I} \rightarrow \mathbf{S}(a, a\_sorted))\,.$$

Chapter 7

Quicksort

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

Quicksort is another popular in-situ sorting algorithm that is based on the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $a[p..r]$.

- **Divide:** Partition the array $a[p..r]$ around a pivot element into two (possibly empty) subarrays $a[p..k-1]$ and $a[k+1..r]$ such that each element of $a[p..k-1]$ is less than or equal to $a[k]$, which is, in turn, less than or equal to each element of $a[k+1..r]$. Compute the index k as part of this partitioning procedure.

- **Conquer:** Sort the two subarrays $a[p..k-1]$ and $a[k+1..r]$ by recursive calls to Quicksort.

- **Combine:** Since the subarrays are sorted in place, no work is needed to combine them: the entire array $a[p..r]$ is now sorted.

The following procedure implements Quicksort:

Procedure *Quicksort* (*a,p,r*) **is**
  if p<r **then**
    k = Partition(a,p,r)
    Quicksort(a,p,k-1)
    Quicksort(a,k+1,r)

To sort an entire array $a[1..N]$, the initial call is $Quicksort\ (a, 1, N)$, where N is the length of the array.

Quicksort has a running time of $O(N^2)$ in the worst case, but it is typically $O(N \lg N)$. In practical situations, a fine-tuned implementation of Quicksort beats most sorting algorithms, including sorting algorithms whose theoretical complexity is $O(N \lg N)$ in the worst case.

Though in the partitioning phase selection algorithms can be used to pick up good pivots, thus giving a variant with $O(N \lg N)$ worst-case running time, however we did not go into the details of such algorithms in order to analyse them and to elaborate declarative implementations for them.

As we have seen in declarative implementations we need to avoid in situ sorting, and in order to be able to do so we need to require more space. The first solution with pure declarative semantics will have a space complexity linear in time complexity, in the sense that if the algorithm runs in $O(N \lg N)$, then the space required will be $O(N \lg N)$ as well, and so on, whereas the second solution, using space optimization by tail-recursion, will have a space complexity of $O(N)$, the same as the imperative solution's.

7.1 Using a Matrix

Just as we did in the case of Bubblesort, in order to avoid sorting in-situ, and thus avoiding destructive assignments, we will simulate Quicksort using a matrix. We will refer to each line of the matrix via the parameter j at each call of the recursive procedure *Quicksort,* which apart from this and the matrix itself as parameters, is also given the left and right bounds, p and r, and the accumulated parameter a_sorted, in which at the final position of the current pivot, once found, the pivot is placed.

It is worth mentioning that not every element of the matrix will be accessed, and possibly nor each line of it will be accessed at all. That is why we defined the `Mat` as an open array of `Vect` type. All will depend on the time complexity of the recursion, which if it's $O(N \lg N)$, then the space required will be $O(N \lg N)$ as well, and not $O(N^2)$, as we don't need a whole matrix for simulation. This discussion is of course purely theoretic, and it still leaves the question open on the compiler side, how to allocate memory in run-time for such open arrays.

```
TYPE Vect = ARRAY [1..N] OF INTEGER;
     Mat = ARRAY OF Vect;
     Index = ARRAY [0..N] OF [0..N+1];
VAR
     i :  INTEGER;
     a_sorted :  Vect;
     a :  Mat;
```

```
PROCEDURE partitioning (c:Vect; p,r:INTEGER;
                                        VAR b :  Vect; VAR k:  INTEGER);
VAR l:  Index;
BEGIN
  l[p-1] = p-1;
  FOR i:=p TO r-1 DO
     IF c[i] <= c[r] THEN
                        l[i] = l[i-1]+1;
                        b[l[i]] = c[i];
                     ELSE
                        l[i] = l[i-1];
                        b[r+1+l[i]-i] = c[i];
                     END;
  END;
  k = l[r-1]+1;
  c[r] = b[k];
END partitioning;

PROCEDURE quicksort (VAR b:  Mat; j:INTEGER; p,r:  INTEGER;
                                         VAR a_sorted :  Vect);
VAR k :  INTEGER;
BEGIN
  IF p<r THEN
      partitioning (b[j],p,r,b[j+1],k);
      b[j,r] = a_sorted[k];
      quicksort (b,j+1,p,k-1,a_sorted);
      quicksort (b,j+1,k+1,r,a_sorted);
  ELSE
     IF p = r THEN b[j,p] = a_sorted[p];
     END;
  END;
END quicksort;

BEGIN
  quicksort (a,1,1,N,a_sorted);
END.
```

The procedure `partitioning` for the input $c[p..r]$ will always choose as pivot the element $c[r]$, and will return in $b[p..r]$ the rearranged elements of $c[p..r]$ around this pivot which will have position k in $b[p..r]$. Basically the procedure works as follows: it parses $c[p..r]$, at each step comparing the current element with the pivot, and depending on this comparison the current element will be placed either to the beginning or the end of $b[p..r]$, at the position calculated with the help of the index counter l. We require here that the actual parameters p and r should satisfy $p \leq r$, which in our case is always achieved as in `quicksort` the procedure `partitioning` is called only for parameters p and r such that $p < r$.

The input array to be sorted is the first line of the global matrix a, which is given as actual parameter in the first call of `quicksort`, also indicating that we start with the first line of the matrix, and boundaries: 1 and N.

The following two matrices show how the implementation works on two different arrays. In each case the array to be sorted is given in $a[1]$.

$a[1]$:	100	-5	-58	5	5	3	9	25	-10	1000
$a[2]$:	100	-5	-58	5	5	3	9	25	-10	1000
$a[3]$:	-58	-10	25	9	3	5	5	-5	100	
$a[4]$:			25	9	3	5	5	-5	100	
$a[5]$:			-5	5	5	3	9	25		
$a[6]$:				5	5	3	9	25		
$a[7]$:				5	5	3	9			
$a[8]$:				3	5	5				
$a[9]$:					5	5				
$a[10]$:										
a_sorted:	-58	-10	-5	3	5	5	9	25	100	1000

Or as a second example:

$a[1]$:	8	-5	-58	50	5	13	19	25	1	30
$a[2]$:	8	-5	-58	5	13	19	25	1	30	50
$a[3]$:	-5	-58	1	25	19	13	5	8		
$a[4]$:	-58	-5		5	8	13	19	25		
$a[5]$:						13	19	25		
$a[6]$:						13	19			
$a[7]$:										
$a[8]$:										
$a[9]$:										
$a[10]$:										
a_sorted:	-58	-5	1	5	8	13	19	25	30	50

The blank positions of the matrix illustrate well that the space complexity grows linear with the time complexity.

The implementation has a pure declarative semantics, its declarative in-

terpretation being the formula:

$$
\begin{aligned}
\mathbf{I} =_{def}\ & P(c,p,r,b,k) \leftrightarrow \exists\, l : \text{Index} \\
& \quad l[p-1] = p-1 \\
& \quad \wedge \\
& \quad \forall i \in [p\,..\,r-1] \qquad (7.1)\\
& \qquad (c[i] \leq c[r] \ \wedge\ l[i] = l[i-1]+1 \ \wedge\ b[l[i]] = c[i]) \ \vee \\
& \qquad (c[i] > c[r] \ \wedge\ l[i] = l[i-1] \ \wedge\ b[r+1+l[i]-i] = c[i]) \\
& \quad \wedge \\
& \quad k = l[r-1]+1 \qquad (7.2)\\
& \quad \wedge \\
& \quad c[r] = b[k] \qquad (7.3)\\
& \wedge \\
& \ Q(b,j,p,r,a\_sorted) \leftrightarrow \exists\, k \qquad (7.4)\\
& \qquad p < r \ \wedge\ P(b[j],p,r,b[j+1],k) \wedge \\
& \qquad \wedge\ b[j,r] = a\_sorted[k] \wedge \qquad (7.5)\\
& \qquad \wedge\ Q(b,j+1,p,k-1,a\_sorted) \ \wedge\ Q(b,j+1,k+1,r,a\_sorted)] \\
& \quad \vee \\
& \qquad p \geq r \ \wedge\ (p = r \rightarrow b[j,p] = a\_sorted[p]) \\
& \wedge \\
& \ Q(a,1,1,N,a\_sorted)
\end{aligned}
$$

We have to prove that:

$$\forall a\colon \text{Mat}, a\_sorted : \text{ Vect} \quad (\mathbf{I} \rightarrow \mathbf{S}(a[1], a\_sorted))$$

where **S** is the specification for sorting defined in (6.1).

Lemma 7.1.1. Assuming **I**, this implies the following:

$$
\begin{aligned}
P(c,p,r,b,k) \rightarrow\ & \text{if } p \leq r \text{ then} \\
& \quad \mathbf{a}) \ p \leq k \leq r \text{ and} \\
& \quad \mathbf{b}) \ b[p..r] \text{ is a permutation of } c[p..r] \text{ s.t.} \\
& \qquad b[p..\,k-1] \leq b[k] < b[k+1\,..r]
\end{aligned}
$$

Proof:

a) Using (7.1), under the assumption that $p \leq r$, by straightforward induction it can be proved easily the following:

$$P(c,p,r,b,k) \rightarrow \exists\, l : \text{Index} \ \ (\forall i \in [p\,..\,r-1] \ \ p-1 \leq l[i] \leq i)$$

Then for $i = r-1$ we have:

$$P(c,p,r,b,k) \rightarrow \exists\, l : \text{Index} \quad (p-1 \leq l[r-1] \leq r-1)$$

which by (7.2) results $p \leq k \leq r$.

b) Using (7.1), under the assumption that $p \leq r$, by straightforward induction it can be proved:

$$\begin{aligned} P(c,p,r,b,k) \;\rightarrow\; & \exists\, l : \text{Index} : \forall i \in [p\,..\,r-1] \text{ the series} \\ & b[p], ..., b[l[i]], b[r+1+l[i]-i], .., b[r] \textit{ is a permutation} \\ & \text{of } c[p..i] \text{ s.t. } b[p..\,l[i]] \leq c[r] < b[r+1+l[i]-i\,..r] \end{aligned}$$

Then for $i = r-1$ and by (7.2), (7.3) we conclude b). □

Lemma 7.1.2. Assuming **I**, this implies the following:

$$Q(b,j,p,r,a\_sorted) \;\rightarrow\; a\_sorted[p..r] \text{ is a sorted permutation of } b[j,p..r]$$

Proof: We follow induction on the length of the segment: $r-p+1$.

Base case: $r-p+1 = 1$, that is $r = p$

Then we have:

$$\begin{aligned} Q(b,j,p,p,a\_sorted) \;\leftrightarrow\; & \exists\, k \\ & [...false...] \vee \\ & p \geq p \wedge (p = p) \rightarrow b[j,p] = a\_sorted[p] \end{aligned}$$

thus the base case holds.

Induction step:

Hypothesis: We presume it is true for any segment of $b[j]$ of length at most k, that is, for $r-p+1 \leq k < N$.

We prove that the property holds for any segment of length $k+1$, that is, for $r-p+1 = k+1$.

Then it is the case that $p < r$, therefore the first branch of the disjunction will be applied in (7.4):

$$\begin{aligned} Q(b,j,p,p,a\_sorted) \;\rightarrow\; & \exists\, k : \\ & p < r \wedge P(b[j],p,r,b[j+1],k) \wedge \\ & b[j,r] = a\_sorted[k] \wedge \\ & Q(b,j+1,p,k-1,a\_sorted) \wedge \\ & Q(b,j+1,k+1,r,a\_sorted) \end{aligned}$$

which by Lemma 7.1.1.b), hypothesis, (7.3) and (7.5) leads to:

$$\begin{aligned} & b[j+1,p..\,k-1] \leq b[j+1,k] < b[j+1,k+1\,..r] \wedge \\ & b[j+1,p..r] \text{ is a permutation of } b[j,p..r] \wedge \\ & b[j+1,k] = a\_sorted[k] \wedge \\ & a\_sorted[p..\,k-1] \text{ is a sorted permutation of } b[j+1,p..k-1] \wedge \\ & a\_sorted[k+1\,..r] \text{ is a sorted permutation of } b[j+1,k+1\,..r] \end{aligned}$$

which results that $a\_sorted[p..r]$ is a sorted permutation of $b[j, p..r]$.
Thus the induction step holds.
The implication is true for any segment $[p..r]$. □

Corollary 7.1.3. Assuming **I**, this implies the following:

$$Q(a, 1, 1, N, a\_sorted) \rightarrow a\_sorted[1..N] \text{ is a sorted permutation of } a[1, 1..N]$$

Proof: Directly from Lemma 7.1.2. with $b = a$, $j = 1$, $p = 1$, $r = N$. □

Thus we have proved that:

$$\forall a\text{: Mat}, a\_sorted : \text{ Vect} \quad (\mathbf{I} \rightarrow \mathbf{S}(a[1], a\_sorted)) .$$

7.2 Tail-recursive Quicksort

The general Quicksort algorithm (both the imperative and declarative version) contains two recursive calls to itself. After the call to `partitioning`, the left subarray is recursively sorted and then the right subarray is recursively sorted. Of these two calls, the second recursive call is a tail-recursive call, which in the case of the imperative solution, according to [6], could be simulated by the compiler using an iterative control structure. The following version of the imperative solution simulates this tail-recursive call:

Procedure *Quicksort'* (*a,p,r*) **is**
  while p<r **do**
    k = *Partition*(*a,p,r*)
    Quicksort'(*a,p,k-1*)
    $p = k + 1$

The so obtained algorithm itself is still not a tail-recursive algorithm for Quicksort. It reduces to some extent the stack space needed, but this doesn't have a significant effect on the overall time nor on the space complexity, on the other hand, because of the state-dependent while-loop, it is far from being obvious to elaborate an efficient declarative solution for it.

However, we aimed to develop a tail-recursive algorithm for Quicksort in the hope that its declarative version would have a reduced space complexity compared to the one in the previous section, as this approach seemed to work well in the experiments of earlier case studies.

First of all we provide here an imperative tail-recursive pseudo-code for Quicksort, followed by a declarative implementation for it.

The basic idea of this algorithm was to somehow combine the two recursive calls into one in a way that the new recursive call would do both jobs at once. In order to be able to keep in track of the left and right boundaries of sub-segments to be sorted during one call (initially the whole array), we maintain

an auxiliary array `aux`, this having the following structure: when $aux[i]$ has a non-negative value, then $a[i\,..aux[i]]$ will be one of the subsegments to be sorted during the actual call. When $aux[i]$ has the value -1, this denotes the position of an earlier already placed in place pivot. In the remaining positions $aux[i]$ has the value 0. At the first call aux is initialized as follows (in case when $N = 8$):

aux	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	8	0	0	0	0	0	0	0

which means, that during the first call the only segment to be sorted is $a[1..8]$, that is the whole array.

After several calls, aux may have the following values:

aux	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	2	0	-1	-1	7	0	0	-1

In this case, the subsegments to be sorted are $a[1..2]$, $a[5..7]$, whereas $a[3]$, $a[4]$ and $a[8]$ are already elements placed in their proper positions with respect to the finally sorted whole array a.

Recursion will stop when no segments to be sorted are found, case in which aux has all its values set to -1:

aux	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	-1	-1	-1	-1	-1	-1	-1	-1

Assuming the existence of the same partitioning procedure as before, the tail-recursive Quicksort has the following pseudo-code:

Procedure *Quicksort_ TR* (*a,aux*) **is**
  $nr.seg = 0$
  for $i = 1$ **to** N **do**
    if $aux[i] \geq i$ **then**
      $nr.seg = nr.seg + 1$
      $left[nr.seg] = i$
  if $nr.seg > 0$ **then**
    for $i = 1$ **to** $nr.seg$ **do**
      $k[i] = Partition(a,left[i],aux[left[i]])$
    $Update\_aux(aux,left,k)$
    $Quicksort\_TR\ (a, aux)$

where *left* is an array to store the left boundaries of the subsegments to be sorted during a call, and k is an other array for storing the new (final) positions of the pivots, one for each subsegment is case.

The reader may easily convince himself that running such an algorithm it would basically make the same steps in transforming a as the original Quick-

sort does, the only difference would be the order of the steps, but the final cost of the summed up steps is the same for both cases. Using terminologies of graph-theory, the original Quicksort is a "depth-first" algorithm, whereas the tail-recursive is "breadth-first". The effect is the same: every element of a, once being chosen as a pivot, finds its proper final place, and it is not being bothered anymore. Of course, *Quicksort_ TR* as a whole will need more time to maintain the auxiliary variable, but this requires at most $O(N)$ during one call, the same as *Partition* does, so the leading magnitude will not be affected by it.

As far as the space complexity is concerned, we cannot economise on the space either with imperative tail-recursive Quicksort: we save space by tail-recursion removal from the stack (at most $O(N)$), but we need to memorise the auxiliary array, which basically "does the work" of the stack.

The following Alma-0 code is a declarative implementation for tail-recursive Quicksort:

```
TYPE Vect = ARRAY [1..N] OF INTEGER;
     Index = ARRAY [0..N] OF [0..N+1];
     Boundaries = ARRAY [1..N+1] OF [-1..N];
     Pivots = ARRAY [1..N] OF INTEGER;
VAR i,j :  INTEGER;
    aux :  Boundaries;
    a,a_sorted :  Vect;

[PROCEDURE partitioning...  ]

PROCEDURE update_aux(aux,left,right :  Boundaries; k :  Pivots;
                                m :  INTEGER; VAR n_aux:  Boundaries);
BEGIN
  FOR i := 1 TO m DO
     IF left[i] < k[i] THEN
          n_aux[left[i]] = k[i]-1
     END;
     FOR j := left[i]+1 TO k[i]-1 DO
          n_aux[j] = 0;
     END;
     n_aux[k[i]] = -1;
     IF k[i] < right[i] THEN
          n_aux[k[i]+1] = right[i]
     END;
     FOR j := k[i]+2 TO right[i] DO
          n_aux[j] = 0;
     END;
  END;
  FOR i :=1 TO N DO
     IF aux[i] = -1 THEN n_aux[i] = -1;
     END;
  END;
END update_aux;
```

```
PROCEDURE quicksort_tr (a:  Vect; aux :  Boundaries; VAR a_sorted :  Vect);
VAR left,right,nr_seg,n_aux :  Boundaries;
    k :  Pivots;
    a_next :  Vect;
BEGIN
    nr_seg[1] = 0;
    FOR i := 1 TO N DO
        IF aux[i] >= i THEN nr_seg[i+1] = nr_seg[i] + 1;
                            left[nr_seg[i+1]] = i;
                            right[nr_seg[i+1]] = aux[i];
                       ELSE nr_seg[i+1] = nr_seg[i];
        END;
    END;

    IF nr_seg[N+1] > 0 THEN
        FOR i :=1 TO nr_seg[N+1] DO
                partitioning(a,left[i],right[i],a_next,k[i]);
        END;
        update_aux(aux,left,right,k,nr_seg[N+1],n_aux);
        FOR i := 1 TO N DO
             IF aux[i] = -1 THEN a_next[i] = a[i];
             END;
        END;
        quicksort_tr(a_next,n_aux,a_sorted);
    ELSE
        FOR i := 1 TO N DO
            a[i] = a_sorted[i];
        END;
    END;
END quicksort_tr;

BEGIN
  FOR i := 2 TO N DO
    aux[i] = 0;
  END;
  aux[1] = N;
  quicksort_tr (a,aux,a_sorted);
END.
```

Procedure `quicksort_tr` has in addition as parameter the accumulated variable *a_sorted*, and as locals: *right*, *n_aux* and *a_next* arrays. The array *right*, for storing the right boundaries of the subsegments to be sorted, has the only purpose to increase readability, whereas arrays *n_aux* and *a_next* are variables for the updated auxiliary and the actual array, respectively, to be passed to the next recursive call.

The following tables illustrate the actual parameters for which `quicksort_tr` is called during the recursion for an array of 8 elements, given as in a (note that *a_sorted* will be accumulated during the calls, getting values only in the last recursive call):

a :	10	7	1	9	8	2	3	5
aux :	8	0	0	0	0	0	0	0

a_next :	1	2	3	5	8	9	7	10
n_aux :	3	0	0	-1	8	0	0	0

a_next :	1	2	3	5	8	9	7	10
n_aux :	2	0	-1	-1	7	0	0	-1

a_next :	1	2	3	5	7	9	8	10
n_aux :	1	-1	-1	-1	-1	7	0	-1

a_next :	1	2	3	5	7	8	9	10
n_aux :	-1	-1	-1	-1	-1	-1	7	-1

a_next :	1	2	3	5	7	8	9	10
n_aux :	-1	-1	-1	-1	-1	-1	-1	-1

In the last call, as no subsegments to be sorted are found, a_sorted will be made equal with a_next from the previous call.

The time complexity of the declarative version has the same magnitude as of the imperative version, that is, when choosing good pivots (as mentioned at the beginning of this chapter) is $O(N \log N)$, whereas with optimized memory usage due to tail-recursion, the space complexity of the declarative tail-recursive Quicksort will be reduced from $O(N \log N)$ to $O(N)$.

We can conclude that there exists a declarative implementation for Quicksort that has the same time and space complexity as the imperative solution. However, due to limited expressivity of declarative programming, the implementation itself manifests a far more complicated code than the original imperative solution, which would result an even more complicated, though achievable logical proof for the implication $\mathbf{I} \rightarrow \mathbf{S}(a, a\_sorted)$.

This is a very critical landmark in declarative programming, as the declarative semantics here does not outweigh the lesser expressivity by offering programs easy to understand, modify or verify, but the other way around. Under this consideration we didn't find it relevant to pursue a detailed correctness proof for this case.

Chapter 8

Conclusions

Throughout this project our attention has been set on two main goals: first, we aimed to extend the computation mechanism introduced by Apt and Bezem [1] and to formulate soundness and completeness results for this extension, and second, to explore declarative programming by examining declarative implementations of non-trivial computational problems, providing formal proofs for the implication $\mathbf{I} \rightarrow \mathbf{S}$.

As far as the achieved theoretical results are concerned, the computation mechanism has been extended for possibly recursive, but consistent procedures. Restricted soundness and completeness results were formulated and proved for implementations where the scope of any negation and/or the premise part of any implication consists of only one non-procedural atom. Soundness and completeness proofs for the general case have been left as a further research topic as, because of both negation and/or recursion may be present, they have to deal with possibly infinite subsidiary trees, this latter causing difficulties in our inductive proof.

The case studies of Chapters 5–7 provide declarative implementations written in the Alma-0 programming language. This experimental language follows the computation mechanism spelled out in Chapter 2., though it doesn't realise it faithfully as we have seen through the examples of some extreme cases. But this shouldn't concern us, as the implementations are safe in this respect, they do not contain negations, and the premise part of the implications are α-closed, so Alma-0 will compute likewise our computation mechanism does. These implementations also satisfy the requirements of the restricted soundness theorem (Theorem 2.3.4.), hence soundness results may apply. Therefore the only task left behind in each case study is the formal proof for the implication $\mathbf{I} \rightarrow \mathbf{S}$.

We find it important to emphasize that these declarative implementations didn't aim "to win" against their imperative counterparts, this would be obviously a naive endeavour in the world of von Neumann computers, but as an experiment it was edifying and instructive to see and analyse their behaviour and performance.

In each of the cases we did elaborate declarative versions of known algorithms (i.e. finding a maximum of an array, Bubblesort and Quicksort) which have the same time complexity as their imperative counterparts, though we had to require more space, hence their space complexity is increased. We have also seen, that in case we can "rephrase" our formulas, so that program parts responsible for increased space complexity are organised in tail-recursive procedures, then by tail-recursion optimisation the space complexity can theoretically be reduced to the the imperative counterparts' space complexity.

This approach seemed to work well in the cases of Maximum element and Bubblesort, where the declarative programs were "bearable" in the sense of readability both as programs and as formulas. In the case of Quicksort, unfortunately, the obtained code became overly complicated. In our view, this is primarily caused by the underlying abstraction from machine state, and secondary because the declarative code simulates an imperative algorithm – maybe it is not the most fortunate enterprise.

Despite these inconveniences, we still consider declarative programming an interesting open field to explore, as it may steal the show in more logic oriented computational problems or some areas of artificial intelligence.

Bibliography

[1] Apt, Krzysztof and Bezem, Marc (1998): Formulas as Programs, in: *The Logic Programming Paradigm.* Springer. Also available under: `http://xxx.lanl.gov/archives/cs` (as cs. LO/9811017)

[2] Apt, K., Brunekreef, J., Partington, V. and Schaerf, A.(1998): *Alma-0. An Imperative Language that Supports Declarative Programming.* ACM Toplas. Also available under: `http://www.cwi.nl/~apt`

[3] Apt, K. and Doets, K: A New Definition of SLDNF-resolution. Also available under: `http://www.cwi.nl/~apt/sldnf.ps`

[4] Backhouse, Roland (2003): *Program Construction. Calculating Implementations from Specifications*. Wiley.

[5] Buchholz, W. (1998): A note on SLDNF-resolution. *Journal of Logic and Computation*, 8(2)

[6] Cormen, Th. H., Leiserson, Ch.E., Rivest, R.L. and Stein, C. (2001): *Introduction to Algorithms.* 2nd Ed. The MIT Press.

[7] Doets, Kees (1994): *From Logic to Logic Programming.* The MIT Press.

[8] Louden, Kenneth C. (1997): *Compiler Construction. Principles and Practice.* PWS Publishing Company.

[9] Partington, V. (1997): *Implementation of an Imperative Programming Language with Backtracking.* Tech. Rep P9712, Departments of Mathematics, Computer Science, Physisc & Astronomy, Univ. of Amsterdam, The Netherlands. Also available under: `http://www.cwi.nl/~apt`

[10] Robinson, J.A. (1965): *A Machine-Oriented Logic Based on the Resolution Principle*. Journal of ACM, Vol 12.

[11] Sebesta, Robert W. (2002): *Concepts of Programming Languages.* 5th Ed. Addison-Wesley.

[12] Sethi, Ravi (1996): *Programming Languages. Concepts and Constructs.* 2nd Ed. Addison-Wesley.

[13] Warren, David H.D (1983): *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, Menlo Park.

[14] Webber, Adam (2002): *Modern Programming Languages. A Practical Introduction.* Franklin, Beedle & Associates, Inc.