university of groningen

faculty of mathematics and natural sciences

Analysis of the Euclidean Feature Transform algorithm

Bachelor thesis - Computing Science

21st August 2012

Student: Sebastian R. Verschoor

Primary supervisor: prof. dr. Gerard R. Renardel de Lavalette

Secondary supervisor: prof. dr. Wim H. Hesselink

Abstract

In this thesis I mechanically verify the correctness and linear time complexity of the core of the Euclidean Feature Transform (EFT) algorithm, using the proof assistant PVS. The EFT algorithm computes the EFT function for a data grid of arbitrary dimension. The EFT function calculates the set of nearest “background” data points, for each data point in the grid. The distance between data points is measured by the standard Euclidean distance.

Contents

1 Introduction **3**

2 Context **4**

2.1 Program Correctness 4

2.1.1 Hoare triples 4

2.1.2 Total correctness while rule 5

2.1.3 Notation 5

2.2 Mechanical verification 5

2.2.1 Concept of mechanical verification 6

2.2.2 Using the computer 6

2.2.3 Conclusions from mechanical verification 6

2.2.4 Mechanically verifying algorithms 7

2.3 PVS 7

2.3.1 PVS properties 8

2.3.2 Proving with PVS 8

2.3.3 programs.pvs 9

3 Euclidean Feature Transform **10**

3.1 Mathematical definition 10

3.1.1 Two dimensional binary EFT 10

3.1.2 Generalising the definition 11

3.1.3 Formal definition 11

3.2 The EFT algorithm 11

3.2.1 Reduction of the dimensions 11

3.2.2 Original algorithm 12

4 Mechanical Verification of the EFT algorithm **14**

4.1 Specification 14

4.1.1 Mathematics 14

4.1.2 Algorithm version 2 15

4.2 Proof 18

4.2.1 Mathematics 18

4.2.2 Algorithm 22

4.2.3 Time complexity 29

5 Conclusion and evaluation **30**

6 Future work **31**

7 Acknowledgement **32**

A PVS Specification 34
A.1 EFT_program.pvs . . . 34
A.2 EFT_program_statements.pvs . . . 39
A.3 EFT.pvs . . . 42
A.4 programs.pvs . . . 47
A.5 ith_element_theory.pvs . . . 49
A.6 more_floor.pvs . . . 50
A.7 max_nat.pvs . . . 51
A.8 auxilCard.pvs . . . 51

Chapter 1

Introduction

In this thesis I analyse an algorithm that computes the Euclidean Feature Transform (EFT). The goal of this project is to mechanically verify that the algorithm by Hesselink [3] correctly calculates the EFT and does so in linear time complexity.

The Feature Transform is an abstract mathematical notion of distances involving digital data. The high level of abstraction gives the algorithm a broad range of applications, for example calculating the Euclidean skeleton of digital images and volume data [4]. On the other hand, this abstraction makes it difficult to understand why the algorithm is correct, or even whether it is correct.

Both the algorithm and a mathematical proof of its correctness are provided in [3]. I do not derive a new algorithm or proof, instead I analyse the existing ones and, ultimately, mechanically verify the correctness of the algorithm.

In chapter 2, I set the context of the project. I describe the theory for proving the correctness of algorithms (Program Correctness), explain the idea of mechanical verification and analyse the proof assistant software tool called PVS.

In chapter 3, I analyse the Euclidean Feature Transform itself. After explaining what the function calculates, I derive a formal mathematical definition. Subsequently, I transform it to a definition that the algorithm of [3] can compute.

In chapter 4, I analyse the given proof and extend it with the details that are required for mechanical verification. The mechanical verification in the PVS proof files follows the outline from this chapter.

I draw conclusions in chapter 5, recommend future work in chapter 6 and state my acknowledgement in chapter 7. For completeness, appendix A contains the complete PVS specification.

Chapter 2

Context

The goal of this project is to prove that a certain algorithm, called the Euclidean Feature Transform algorithm, is correct. For this purpose, I need to be able to make formal mathematical statements about what it means for an algorithm to be correct. To do this, I use the theory of Program Correctness, which I explain in section 2.1.

The concept of mechanical verification is that almost anything in mathematics can be formally verified. Using a formal language like predicate calculus, mathematics can be built up from a limited set of axioms and rules. By using these rules, almost any mathematical statement can be mechanically verified to be true or false. I elaborate on this idea in section 2.2.

The actual verification heavily depends on the characteristics of the used tools. The most important tool is a proof assistant computer program. In this project I use PVS, and I analyse its characteristics in section 2.3.

2.1 Program Correctness

The basic theory on which the proof of this project relies is that of Program Correctness. The idea of Program Correctness is that a program statement changes the state of the executing machine to some other state. These states can be described by mathematics. To change the state, only a limited set of program commands are allowed. Each command has its rules as to how it changes the state, which is described by a Hoare triple.

2.1.1 Hoare triples

A Hoare triple consists of a precondition, a program statement and a postcondition, where both the pre- and postcondition are predicates that describe the machine state. When the precondition is met beforehand, the program statement establishes the postcondition.

Let P, P', Q, Q', R and J be predicates about the machine state, B a predicate about the program variables, S and T be program statements, E be an expression in the program language and x be a program variable. Then the theory provides the following proof rules for Hoare triples.

The skip command does not change the state:

$$\{P\}\ \mathit{skip}\ \{P\}$$

The assignment command ($\leftarrow$) rewrites variable x with expression E:

$$\{P[E/x]\}\ x \leftarrow E\ \{P\}$$

Sequential composition (^)[1] couples two statements together:

$$\frac{\{P\}\ S\ \{Q\}, \quad \{Q\}\ T\ \{R\}}{\{P\}\ S\hat{}T\ \{R\}}$$

[1] The standard symbol ';' is not available as PVS operator, therefore it is replaced with '^'.

Branching is described by the if-then-else rule:

$$\frac{\{P \wedge B\}\ S\ \{Q\}, \quad \{P \wedge \neg B\}\ T\ \{Q\}}{\{P\}\ \text{if}\ B\ \text{then}\ S\ \text{else}\ T\ \text{end if}\ \{Q\}}$$

Repetition is described by the (conditional correctness) while rule:

$$\frac{\{J \wedge B\}\ S\ \{J\}}{\{J\}\ \text{while}\ B\ \text{do}\ S\ \text{end while}\ \{J \wedge \neg B\}}$$

But the most important rule is the consequence rule. By applying this rule before and after the statements, mathematical theorems can be applied to the proof. In many algorithms, the difficulty of proving the correctness is not to find the correct program statements, but to find the correct application of the consequence rule:

$$\frac{P' \Rightarrow P, \quad \{P\}\ S\ \{Q\}, \quad Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

2.1.2 Total correctness while rule

The above while rule guarantees the conditional correctness of a program, meaning that *if* it will terminate, the postcondition will hold. What remains is to prove that it terminates. An additional function is introduced: the variant function vf. Initially $vf \geq 0$. Furthermore, each execution of the while body needs to lower the function value. The result is the (total correctness) while rule:

$$\frac{\{J \wedge B \wedge vf = V\}\ S\ \{J \wedge vf < V\}, \quad J \wedge B \Rightarrow vf \geq 0}{\{J\}\ \text{while}\ B\ \text{do}\ S\ \text{end while}\ \{J \wedge \neg B\}}$$

Time complexity

The variant function gives information about the time complexity of the while loop. Because vf decreases each time the body is executed, the total number of repetitions of the body can be no more than the initial value of vf. The order of time complexity can be derived directly from the vf if the body is $\mathcal{O}(1)$. In case a statement inside the body has a greater order, the time complexity of the loop is the multiplication of the loop-order and the statement-order.

However, in the case of the EFT algorithm, which has nested loops, multiplication is not enough to prove linear time complexity. A method is applied where the vf of the inner- and outer loop are equal and the vf decreases in both loops. Then, the complexity of the outer loop can be derived directly from the vf.

2.1.3 Notation

The notation of a program correctness proof resembles the above notation. The program statements are numbered lines, the curly brackets, '{' and '}', surround the state predicates and if a consequence rule is not obvious, additional comment is added between '(*' and '*)'.

2.2 Mechanical verification

As stated before, a formal system is built up from a small set of axioms and rules, an example being the predicate calculus. Mechanical verification checks whether a mathematical sentence could have been built up from the set of axioms, using the available rules. This section explains the advantage of building up mathematics using a formal system, it shows how the computer can assist with mechanical verification and finally it discusses which conclusions can be drawn from mechanical verification.

2.2.1 Concept of mechanical verification

Mechanical verification is the ultimate extension of formalising mathematics. The foundation of this process was laid by the ancient Greeks, a well-known example being *Euclid's Elements*. In this work, Euclid formulated five axioms from which he derived a rich set of theorems, using only logical derivations from the axioms and already proven theorems.

The problem with this method of proving mathematics is that a theorem and its proof grow complex and unmanageable very fast. To solve this problem, mathematicians skip many logical steps in a proof that are considered to be trivial. Although this usually results in a proof that is much more understandable, there is also a lot more room for errors. A classical fallacy is the following proof, deriving $2 = 1$ from the assumption $a = b$:

$$\begin{aligned} a &= b \\ a^2 &= ab \\ 2a^2 &= a^2 + ab \\ 2a^2 - 2ab &= a^2 + ab - 2ab \\ 2a^2 - 2ab &= a^2 - ab \\ 2(a^2 - ab) &= 1(a^2 - ab) \\ 2 &= 1 \end{aligned}$$

The problem lies in dividing by the term $a^2 - ab$ in the last step, because $a^2 - ab = 0$. A mechanical verification would require for every division the additional proof that the denominator is non-zero.[2]

2.2.2 Using the computer

By using the computer, the complexity that arises from formalising mathematics can be managed. Almost all mathematical statements can be written down in a formal language, such as a predicate calculus. This language can be parsed and interpreted by a computer program, called a proof assistant. The kernel of the proof assistant defines a predicate calculus, in which the user can state well- formed formulas (wffs). These wffs are either true or false, although the goal is usually to state truths. Using the inference rules of the predicate logic, the user must rewrite the formulas to other formulas, until an axiom has been derived.

The proof assistant can help in two ways. First of all, it gives all the proof obligations that occur when using rules or stating wffs, meaning all wffs that need to be true. When all proof obligations have been verified to be true, the proof assistant validates that the wff is indeed true. In this case the proof assistant acts as a bookkeeper.

The other advantage of using software is that there is some automation in inferring rules. Automation can be very useful when the proof uses exhaustive case distinction, but also when the proof contains arithmetic operations. Some propositional assertions can be computed by the proof assistant. The amount of automation varies for different proof assistants.

2.2.3 Conclusions from mechanical verification

An important question remains: "What conclusions can be inferred from having mechanically verified a formula?" Surprisingly, the answer is not immediately: "That the formula is true." The reason is that there remains some room for errors and there is always human interpretation of the result.

Errors

The proof assistant could contain bugs. The predicate calculus is specified in some programming language and is possibly specified wrong. The solution is to keep the kernel of the proof assistant very small, so that there is little room for errors. The automation module of the proof assistant could contain errors

[2] In fact, the EFT proof contains this non-zero requirement for division at TCC `g_TCC1`.

as well, but luckily there is a solution for that problem. By separating the automation module from the kernel, the kernel can actually be used to mechanically verify the automation module.

Another place for errors is only a theoretical one. The mechanical verification is done upon a real machine, with underlying hardware. There could be a hardware failure, leading to the incorrect verification a formula. By executing the verification multiple times, preferably on different machine, the chances of this happening are negligible.

Interpreting the result

The formula could be stated wrong, which means that the wrong formula has been verified to be true. There is always the part where a human must interpret the result, and humans make mistakes. By keeping the formula as simple as possible, the room for interpretation is minimal, which is the best that can be achieved. There is no point in trying to verify that interpretation of the formula is the correct one, because this would require a new specification for correctness, which needs interpretation itself.

Value of mechanical verification

With all this room for errors and interpretation, one might believe that there is not much additional value in a mechanical verification compared to mathematical proofs. But there are many benefits, because mechanical verification eliminates many possible errors in a proof, but does not introduce any errors. Program bugs and hardware failures are comparable to typing- and printing errors. There is always interpretation of a mathematical statement, so this is not an issue introduced by mechanical verification. The conclusions that can be drawn from mechanically verifying a specific problem depend most heavily on two factors: how simple is the problem specified and which proof assistant was used?

2.2.4 Mechanically verifying algorithms

Mechanically verifying an algorithm is not any different from verifying mathematics, in theory. Using the Hoare triples from program correctness, a program can be described by a mathematical statement. Yet there are some minor differences.

The first difference is the specification of the Hoare triples in the proof assistant. The specification itself could contain bugs. Now there is room for errors in both the specification of the states and in the specification of Hoare triples. The solution is to keep the specification as simple as possible.

The interpretation of the program statements themselves is difficult. The specification of statements could be wrong, meaning that the wrong algorithm is being verified. If the result is an executable algorithm with a Hoare triple that has the same pre- and postcondition, then there is no real harm done. But one must be careful when drawing any conclusions about the algorithm.

An actual implementation of the algorithm is a whole other problem. It can contain typing errors or other bugs, depend upon the language in which it was implemented, possibly depend on the compiler that was used, etcetera. One important assumption underlying the theory of Program Correctness, is that the state is not altered in between or during statements, except by the specified commands. In concurrent machines, this assumption is often incorrect. Therefore, there is a big difference between the correctness of the algorithm and the correct execution of a computer program.

2.3 PVS

PVS stands for Prototype Verification System. It was developed by the Computer Science Laboratory of Stanford Research Institute (SRI) International. I chose PVS based on the advice of my supervisor, Wim Hesselink, because he has the most extensive experience with this proof assistant.

As explained in section 2.2.3, the properties of the proof assistant are very important for drawing conclusions of mechanical verification, so these properties are examined in this section. To get an idea of working with a proof assistant, the interface of PVS is explained. Finally, the implementation of the Program Correctness theory in PVS is analysed.

2.3.1 PVS properties

One can say something about the quality of PVS as a proof assistant, by looking at some of its properties. On the one hand it should be user friendly, making specification of problems easy and interpretation of those specifications straightforward. On the other hand, the underlying code of the program should be small and verified where possible, so that there is very little room for errors, which makes it possible to draw strong conclusions.

PVS is quite user friendly, although there is a steep learning curve. PVS notations are very close to standard mathematical notation, so that a PVS specification can easily be checked by a mathematician.

On the other hand, the conclusions that can be drawn from verification with PVS are not very strong. PVS has quite a large kernel and its proof automation is not verified. This means PVS has a lot of room for errors, but not necessarily that it contains errors. Even if it did, that does not directly mean that such an error affects the proof of this project. There is a world wide community of PVS users that send in bug reports regularly. These reports are treated, leading to improved versions of PVS. Most of the bugs reported are not about the soundness of the prover, but about other aspects of its behaviour. Therefore the implicit assumption of this project, that PVS contains no significant errors, is a reasonable one.

2.3.2 Proving with PVS

A PVS proof consists of two major parts. First, there is the specification language. PVS uses a typed higher order logic in which the user can formally specify the problem, using definitions and theorems. The other part is the prover, in which the specified theorems can be rewritten to logically equivalent statements, until they are considered true.

PVS Specification Language

In the specification language, the user specifies the mathematics in formal logical sentences. First, the user defines everything to be used in the proof: constants, variables, functions, sets, etcetera. Then the user states theorems and lemmas, which are logical sentences that can be either true or false.

The specification file is the most important tool for interpreting the resulting proof. When the problem can be stated in a clear sentence, using only simple definitions, it is clear what is actually being proven and there is very little room for errors.

Each of the definitions have a type, which possibly inherits from other types. Because of the types, PVS derives Type Correctness Conditions (TCCs) from definitions. In addition to the correctness of the main theorem, proof obligations are generated to prove the correctness of all TCCs.

A much occurring example of a TCC is the non-negativity of natural numbers. Assume the user defines a function $f : (\mathbb{N}^2 \to \mathbb{N})$ as $f(a, b) = a - b$. This definition would only result in a natural number if $a \geq b$, which can not be derived from the current context. Two solutions are possible: change the type to $f : (\mathbb{N}^2 \to \mathbb{Z})$, or add the context to the definition: $f(a, b) = (a \geq b \ ?\ a - b : 0)$. The first solution has the problem that the result of f cannot be used directly as a natural number, so this possibly introduces many TCCs in other definitions where f is used. The second definition has the problem that the user might inadvertently use f when $a < b$ and still get a result, which is incorrect.

There is no best solution, it is the user who must decide which one to use, depending on the context of the definition in the specification. The TCCs are a very tricky part of specifying a problem, which results in some non-straightforward definitions.

PVS Prover

Once the problem has been stated in theorems, the user can try and prove that the given theorems are correct. For each theorem, the user starts the prover, which gives a proof obligation that always has the same form[3]:

$$P_0 \wedge P_1 \wedge \ldots \wedge P_m \Rightarrow Q_0 \vee Q_1 \vee \ldots \vee Q_n \tag{2.1}$$

[3] In the prover the notation is vertical, where a horizontal line represents the '⇒', but the meaning is the same.

In order to prove a theorem, it needs to be rewritten with the logical rules that PVS provides. These rules rewrite a wff to a logically equivalent one, until the whole rule is equivalent to an axiom or the wff *true*.

An example rule rewrites (2.1) according to the following equivalence:

$$P \Rightarrow Q \equiv \neg P \vee Q$$

P representing the conjunction of P_i, Q the disjunction of Q_j, with $i \in [0..m)$ and $j \in [0..n)$. The consequence is that a wff is considered to be true when any of P_i is false, or any of Q_j is true, or any P_i is equal to any Q_j. This applies directly to standard form of PVS proof obligations.

Now a proof obligation ($P \Rightarrow Q$) could occur where Q is of the form $Q_0 \wedge Q_1$. Assuming P, both Q_0 and Q_1 need to be proven. Using the "split" rule, PVS breaks the proof down in two smaller proof obligations: $P \Rightarrow Q_0$ and $P \Rightarrow Q_1$. Only if both obligations have been proven, the original theorem is proven. Breaking up proofs like this is so common, that PVS provides another tool for keeping track of all branches in a proof: the proof tree. There is no functionality in the proof tree, but it is essential for the user to keep an overview of the proof status.

When a theorem has been proven, PVS stores the proof, so there is no need to prove it again. A proven theorem is not yet complete, as the proof could rely on auxiliary lemmas. Only when these lemmas have all been proven, the theorem is considered complete. The advantage of using unproven lemmas is that it can greatly simplify a proof, leaving the proof obligation of the lemma for a later time.

The proof of a theorem could depend upon lemmas, which in their turn could depend on other lemmas, creating a large dependency tree of proofs. In a sense, the specification is no different from a prover session, which is also a tree of proofs. The only difference is that the rules have been made explicit, so that the proof becomes manageable.

Automation

PVS has a small amount of automation. It can do some basic arithmetic operations and some very basic rewriting of logical operators, but no more than that. PVS has strategies, which just chains several basic rules behind each other. Strategies can be used just like regular rules. Users can specify their own strategies, which potentially heighten the amount of automation in the proof. No custom strategies have been used in this project.

2.3.3 programs.pvs

In order to prove algorithms with PVS, an implementation of the Program Correctness theory is required. This is provided by the file programs.pvs, which contains:

- The definitions for program statements (if-then-else, composition etc.)
- The definitions for total and conditional correctness
- Useful proven lemmas, stating properties of statements and their correctness

The most important definition is that of `tcHoare`: it states that if the precondition is true before the program is run, the program both terminates and the postcondition is true after the program is run.

The programs theory is just what one would expect, but for a small detail. The theory distinguishes between a `program` and a `command`, the difference being that commands always terminate. A command can be lifted to a program (with the `lift` function) and by doing so, the user creates a program that will terminate. A while-program obviously has no equivalent command, so termination of a while-program needs to be proven explicitly, usually by applying the `whileTheorem`.

The implementation of the Program Correctness theory is small and straightforward, therefore it can be considered to be correct.

Chapter 3

Euclidean Feature Transform

In this chapter, I will analyse the Euclidean Feature Transform (EFT). In section 3.1, I explain what the EFT is and give formal definition. For a long time, it was thought that the EFT could not be computed in linear time and had to be approximated by the chamfer distance [1]. In section 3.2, I analyse the definition and rewrite it into one that can be computed in linear time. The linear time algorithm is fully stated in the same section.

3.1 Mathematical definition

Before any analysis of the EFT can be done, the exact meaning will have to be formulated. This section does that, starting with a simple example and inferring a general definition from there.

3.1.1 Two dimensional binary EFT

The EFT can be explained easiest by an example:

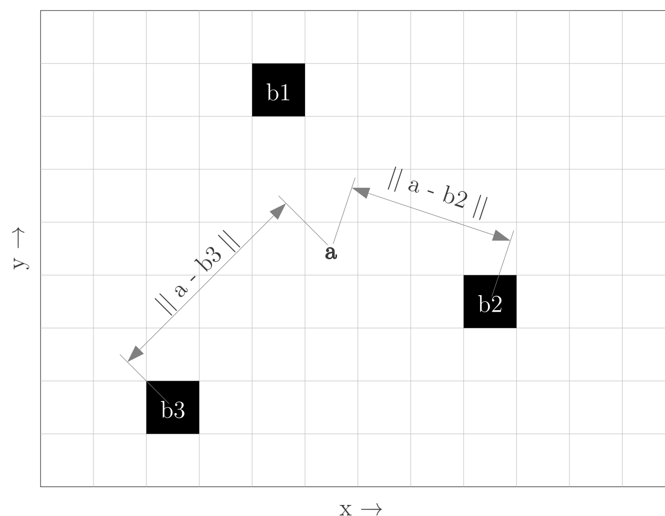

Figure 3.1: Binary image with background pixels

Figure 3.1 represents a binary image A (with $A \subseteq \mathbb{N}^2$) and each square represents a pixel. The black pixels ($b1$, $b2$ and $b3$) are elements of the background pixel set, denoted by B (with $B \subseteq A$). For every $a \in A$, the distance to the nearest background pixel $b \in B$ is called the distance transform of a, or $dt(a)$. In case of the Euclidean distance transform (edt), the distance is measured by using the Pythagorean formula. In the example: $edt(a) = ||a - b1|| = ||a - b2|| = \sqrt{(a.x - b1.x)^2 + (a.y - b1.y)^2}$ $(= \sqrt{10})$.

Feature transform is an abbreviation of nearest feature transform. The Euclidean Feature Transform (EFT) is a function that calculates a set of nearest background pixels for each pixel. More formally: $EFT : A \rightarrow 2^B$ and $b \in EFT(a)$ iff $||a - b|| = edt(a)$. In the example: $EFT(a) = \{b1, b2\}$, because $edt(a) = ||a - b1|| = ||a - b2||$. But $b3 \notin EFT(a)$, because $||a - b3|| > edt(a)$. Note that for calculating the EFT, the calculation of the square root is superfluous, because $\sqrt{x} = \sqrt{y} \equiv x = y$[1].

3.1.2 Generalising the definition

The above definition works for two-dimensional binary images, but can be extended to a broader range of datasets. Let d be the dimension, then the data is represented by a rectangular box A, where $A \subseteq \mathbb{N}^d$ is a subspace of the standard Euclidean vector space $\mathbb{R}^d$. For grid points x and p, the squared distance is $||x - p||^2 = \sum_i^d (x_i - p_i)^2$.

In order to address more than binary images, the grey-level function h is introduced ($h : A \rightarrow \mathbb{R}$), which is added to the squared distance. This gives the new "distance" function $f : A^2 \times (A \rightarrow \mathbb{R}) \rightarrow \mathbb{R}$:

$$f(x, p, h) = ||x - p||^2 + h(p) \tag{3.1}$$

Note that there is no distinction any more between foreground and background. Because h is defined for all data-points in A, B is made equal to A. There is an instance of h that is equivalent with the binary case: $h(p) = (p \in B \ ?\ 0 : \infty)^2$, so introducing h is indeed a generalization of the original problem.

By adding h to the definition, the multi-dimensional problem can be reduced to the one-dimensional problem. This reduction is a very important step in development of the algorithm, so the details will be explained in section 3.2.1.

3.1.3 Formal definition

The generalised distance function (3.1) is used for the definition of edt (with $edt : A \times (A \rightarrow \mathbb{R}) \rightarrow \mathbb{R}$):

$$edt(x, h) = Min\{f(x, p, h) \mid p \in A\} \tag{3.2}$$

The definition of edt leads to the definition of EFT (with $EFT : A \times (A \rightarrow \mathbb{R}) \rightarrow 2^A$)

$$EFT(x, h) = \{p \in A \mid f(x, p, h) = edt(x, h)\} \tag{3.3}$$

Note that both definitions do not carry any explicit information about the dimension. Implicitly, the dimension is embedded in the types of the variables.

3.2 The EFT algorithm

Both the algorithm and the mathematical proof are adapted from [3]. The important step of the proof is the reduction on the dimensions, which is explained in section 3.2.1. The algorithm has some elegant properties, which are examined in section 3.2.2.

3.2.1 Reduction of the dimensions

The reduction of the higher dimensional EFT problem to the one dimensional EFT problem is done by induction on the dimensions. By giving a recursive formula for the edt, a recursive formula for EFT can be found. Assuming a solution is available for solving the one-dimensional case, all that remains is to find a solution that solves the inductive step: computing the EFT in dimension $d > 1$, using the EFT in dimension $d - 1$.

One line of data is extracted from the rectangular dataset A. Let $A \subseteq \mathbb{N}^d$ be the Cartesian product of the form $A = A' \times [0..n)$, where $A' \subseteq \mathbb{N}^{d-1}$ and $n \in \mathbb{N}^+$. For every $y \in A'$, $edt(x, h)$ (and eventually $EFT(x, h)$) can be computed for all grid points $x = (y, z)$, where $z \in [0..n)$:

$$edt(x, h) = edt((y, z), h) = Min\{||(y, z) - (p, q)||^2 + h(p, q) \mid q \in [0..n), p \in A'\}$$

[1] Assuming x and y are both ≥ 0, which is true when they represent distances.

[2] A practical implementation might replace ∞ with a value W (with $W > diagonal^2$). In this case, $diagonal$ is the length of the diagonal of the grid, so that W is bigger than the largest distance inside the grid.

Let y be fixed, while z varies over the range $[0..n)$. With the Theorem of Pythagoras, the definition is split in two parts, giving the following formula:

$$edt((y,z),h) = Min\{(z-q)^2 + Min\{||y-p||^2 + h(p,q) \mid p \in A'\} \mid q \in [0..n)\}$$

The inner minimum is actually the distance transform in dimension $d-1$. By the inductive hypothesis, this can be computed. Replacing the inner distance transform by the function h' gives the following formula:

$$edt((y,z),h) = Min\{(z-q)^2 + h'(q) \mid q \in [0..n)\}$$

where $h'(q) = Min\{||y-p||^2 + h(p,q) \mid p \in A'\}$. Again a familiar form can be recognised: the new formula is a one dimensional distance transform, with the new function h'. So, both the base case ($d = 1$) and the inductive step ($d > 1$) can be computed by computing the one dimensional case.

A minor detail for implementing this is the data-structure. The assumption is that the case $d = 1$ is solvable, when the data points are integers. In terms of equation (3.2): $p \in A \equiv p \in [0..n)$. But for different dimensions, the data could be of different size. Therefore, we need to add n as a parameter to the algorithm, so that the function h over interval $[0..n)$ represents the data line. Renaming the variables (z with x and p with q) gives the final formula:

$$edt(x,n,h) = Min\{(x-p)^2 + h(p) \mid p \in [0..n)\} \tag{3.4}$$

The corresponding alteration is required for the EFT:

$$EFT(x,n,h) = \{p \in [0..n) \mid (x-p)^2 + h(p) = edt(x,n,h)\} \tag{3.5}$$

Finding the correct data representations for h' is not trivial and it is considered outside the scope of this project to give the details of higher dimension implementation. A requirement for this implementation is that each element of h' can be found in $\mathcal{O}(1)$. Such implementation exists and [3] even provides an example for three dimensional EFT.

The important conclusion from the above discussion is that it is enough to verify the one dimensional algorithm. Technically, the above reduction could be mechanically verified as well. This is further discussed in chapter 6.

3.2.2 Original algorithm

The complete algorithm, as given by the article, is stated in algorithm 1. There are some interesting properties of this particular implementation that are discussed in this section.

The first thing that might be noticed is the absence of the parameter h in the body of the algorithm. This is actually embedded in the functions f (lines 5 and 25) and g (line 11). Function $f(x,p)$ corresponds to equation 3.1:

$$f(x,p) = ||x-p||^2 + h(p)$$

Function $g(p,q)$ solves the equation $f(x,p) = f(x,q)$, or more precisely: the inequality $f(x,p) \leq f(x,q) \equiv x \leq g(p,q)$. For the EFT, this solution is given by:

$$g(p,q) = \left\lfloor \frac{q^2 - p^2 + h(q) - h(p)}{2(q-p)} \right\rfloor$$

The reason for not making these functions explicit is that they can be changed to compute the Feature Transform for different types of distances. For example, the solutions for the Manhattan distance and the chessboard distance are provided by Meijster et al. [5]. Although a solution exists for a broad class of distances, not all types of distances will necessarily have a solution for g that can be computed in $\mathcal{O}(1)$.

Algorithm 1 Original OneFT

```
1.  procedure OneFT(n : ℕ; h : [0..n) → ℝ)
2.      var k, q : ℤ; w, j, y, y1, p : ℕ; t, at : [0..n] → ℤ; FT : [0..n) → ℙ(ℤ)
3.      q ← 0; t(0) ← 0; at(0) ← 0
4.      for k ← 1 to n - 1 do
5.          while q ≥ 0 ∧ f(t(q), at(q)) > f(t(q), k) do
6.              q ← q - 1
7.          end while
8.          if q < 0 then
9.              q ← 0; at(0) ← k
10.         else
11.             w ← 1 + g(at(q), k)
12.             if w < n then
13.                 q ← q + 1
14.                 t(q) ← w; at(q) ← k
15.             end if
16.         end if
17.     end for
18.     t(q + 1) ← n; at(q + 1) ← n - 1
19.     for j ← 0 to q do
20.         y1 ← t(j + 1) - 1
21.         for y ← t(j) to y1 do
22.             FT(y) ← {at(j)}
23.         end for
24.         for p ← at(j) + 1 to at(j + 1) do
25.             if f(y1, p) = f(y1, at(j)) then
26.                 FT(y1) ← FT(y1) ∪ {p}
27.             end if
28.         end for
29.     end for
30.     return FT
31. end procedure
```

Closer inspection of the algorithm reveals that it actually consists of two phases. The first phase (lines 3-17) is called the build phase, the second phase (lines 18-29) is called the harvest phase. By adjusting only the harvest phase, it is possible to compute the *simple EFT* or the Euclidean distance transform *edt*. The simple EFT computes just one element of the EFT set per data point and can be computed by removing the second inner loop (lines 24-28) from the harvest phase. The edt can be computed by removing the same loop and replacing line 22 with “dt(y) ← f(y, at(j))”.

Existing proof

The other reason for choosing this particular implementation is that it comes from an article with a very formal proof, whereas other proofs rely on geometric arguments [5] or miss argumentation regarding the separation of the two phases [2]. The proof that this thesis analyses is exactly the same as the proof in [3], only with more detail.

Chapter 4

Mechanical Verification of the EFT algorithm

In this chapter I give more detail regarding the mechanical verification of the EFT algorithm. Since the specification of the problem is subject to human interpretation, in section 4.1, I discuss every detail of it and argue why it is correct. In section 4.2, I give the details of the PVS verification.

4.1 Specification

As explained in section 2.2.3, the specification of the problem can not be verified and is subject to interpretation. In this section I will pass each part of the specification and argue why it is correct.

Since the ultimate goal of the mechanical verification is to prove one theorem, I will split out all the details of this theorem:

EFT_program.pvs

```
370 program_correct_oneft: THEOREM
371   tcHoare(fullset[state], program_oneft, Q_oft)
```

The surrounding predicate `tcHoare` is part of the programs theory. This theory was already provided and I assume it to be correct, as explained in section 2.3.3.

In section 4.1.1, I discuss the precondition and postcondition of the algorithm and in section 4.1.2, I discuss the program statements.

4.1.1 Mathematics

The precondition is not a very interesting one: the set `fullset[state]` is by definition the whole state space, meaning that it does not matter what the state is before executing `program_oneft`.

The postcondition is a lot more interesting, as it contains the definition of the EFT: $Q_{oft} \equiv Q_{fth} \equiv \forall(x \in \mathbb{N}) : x < n \Rightarrow EFT'(x) = EFT(x, n, h)$.[1] This predicate states that the EFT has been calculated and stored in variable EFT', for each $x \in [0..n)$.

What remains is the exact definition of EFT, which should be equivalent to the equations (3.4) and (3.5). The specification is kept as close as possible to the mathematics:

EFT.pvs

```
58 % One-dimensional squared Euclidean distance
59 dist(x, p): nat =
60   (x - p) * (x - p)
61
62 % Distance, including h
63 f(x, p, h): nat =
64   dist(x, p) + h(p)
65
66 % Nearest distance to any point
```

[1] The notation, with the accents, is explained in section 4.1.2.

```
67 edt(x, n, h): nat =
68   min({ d | EXISTS p: d = f(x, p, h) AND p < n })
69
70 % All points with the nearest distance
71 EFT(x, n, h): setof[nat] =
72   { p | p < n AND f(x, p, h) = edt(x, n, h) }
```

There are subtle differences in the set-notation, but these only move the predicates left of the '|' symbol to the right side, because this is the only formally correct set notation in PVS. The variable `d` is introduced in combination with the existential quantifier $\exists$ in the definition of `edt`, to store the value of `f(x, p, h)`. Note that the definition of f is the one dimensional implementation of equation (3.1).

Types

The types differ slightly from the mathematical definitions from chapter 3, which has mostly to do with the TCCs generated by PVS, see also section 2.3.2. I list the differences here and explain why I changed them. These changes in variable types are also reflected by the function types.

h New type is $\mathbb{N} \to \mathbb{N}$, not $[0..n) \to \mathbb{R}$. Since n is a variable, it can not be used in type declaration $[0..n)$, so I used the next closest type: $\mathbb{N}$. The return type should just have been $\mathbb{R}$, but $\mathbb{N}$ was an unfortunate choice made in the beginning of the project. Changing it at this stage gives some unsolvable TCCs and requires some proofs to be redone, which is too much work for now.

p, x New type is $\mathbb{N}$, not $[0..n)$. This applies to all variables that represent data points. Again, the reason is that n is a variable and can not be used in a type declaration.

d Now of type $\mathbb{N}$, not $\mathbb{R}$. Since the return type of h is $\mathbb{N}$, the distance (including h) is always a natural number.

All type changes are small and I believe they do not affect the correctness of the specification. The specification of the mathematics itself is very small and is syntactically very similar to the mathematical definitions, leaving very little room for errors.

Variables h and n

Both h and n are variables in the theory `EFT`, while both remain constant throughout the whole specification. The same applies to the algorithm variables, where `s‘h` and `s‘n` are parameters to the procedure OneFT, but are not changed inside the procedure. Only the solution for the higher dimension EFT algorithm gives arguments for h and n that are variable. I did not fix this problem in this project, but I recommend it as future work, see also chapter 6.

To see that this is a problem, consider the program "$n \leftarrow 1 \wedge EFT'(0) \leftarrow \{0\}$". This program fulfils the postcondition, but it is not a satisfying solution to the problem. A simple solution would be to add information to the precondition and postcondition, stating that `s‘h` and `s‘n` remain constant.

A better implementation would be to declare h and n as constants or as parameters of the theory `EFT`. Both variables `s‘h` and `s‘n` should be removed from the state variable `s` and replaced by the constant of the `EFT` theory. The parameter solution has the advantage that it can be used in a proof of the higher order EFT algorithm.

For the remainder of this thesis, I will explicitly state h and n as parameters of the definitions, to ensure correspondence with the PVS specification. To understand the definitions, it might help to ignore these parameters since they remain constant at all times.

4.1.2 Algorithm version 2

Algorithm 1 is not yet ready, some adjustments are required to be able to verify it. The fully adjusted algorithm is given in algorithm 2 and is attached in appendix A.2, which contains the entire PVS specification of `program_oneft`.

Even if algorithm 2 is not the same as algorithm 1, that is not a real problem. It just means I have found an alternative implementation to the same problem. However, I discuss all differences between the algorithms in this section, because the differences are not essential and the adjusted algorithm still represents the original algorithm.

Algorithm 2 Adjusted OneFT (full)

```
1.  procedure OneFT(n : ℕ⁺; h : ℤ → ℕ)
2.      var q', w : ℤ; t', at' : ℤ → ℕ; k : ℕ⁺; j, y, y1, r : ℕ; EFT' : ℕ → ℙ(ℕ)
3.      q' ← 0; t'(0) ← 0; at'(0) ← 0
4.      k ← 1
5.      while k < n do
6.          while q' ≥ 0 ∧ f(t'(q'), at'(q'), h) > f(t'(q'), k, h)  do
7.              q' ← q' - 1
8.          end while
9.          if q' < 0 then
10.             q' ← 0; at'(0) ← k
11.         else
12.             w ← 1 + g(at'(q'), k, h)
13.             if w < n then
14.                 q' ← q' + 1
15.                 t'(q') ← ( w ≥ 0 ? w : 0 ); at'(q') ← k
16.             else
17.                 SKIP
18.             end if
19.         end if
20.         k ← k + 1
21.     end while
22.     t'(q' + 1) ← n; at'(q' + 1) ← n - 1
23.     y ← 0; r ← at'(0) + 1
24.     j ← 0
25.     while j ≤ q' do
26.         y1 ← ( t'(j + 1) - 1 ≥ 0 ? t'(j + 1) - 1 : 0 )
27.         while y ≤ y1 do
28.             EFT'(y) ← {at'(j)}
29.             y ← y + 1
30.         end while
31.         while r ≤ at'(j + 1) do
32.             if f(y1, r, h) = f(y1, at'(j), h) then
33.                 EFT'(y1) ← EFT'(y1) ∪ {r}
34.             else
35.                 SKIP
36.             end if
37.             r ← r + 1
38.         end while
39.         j ← j + 1
40.     end while
41.     return EFT'
42. end procedure
```

Loops

First, the for-loops need to be rewritten to while-loops and the if-statements to if-else-statements. This is because Program Correctness only provides tools to validate these. Besides, for-loops can be seen as syntactic sugar for while- loops.

Renaming

I did some renaming, which has no influence on the functionality. For example, the article uses q as both a mathematical function and a program variable. Although this is a very natural thing to do, because the program variable stores the function-value, in a verified proof the distinction is important. I gave

the program variables an accent to distinguish them: q is the mathematical object, q' is the program variable.

The PVS specification separates between variables and mathematical objects by the file in which they are declared. The file EFT.pvs contains all mathematical definitions, while the file EFT_program.pvs contains a state variable `s`, in which the variables are declared. EFT_program.pvs distinguishes by adding the scope to the parameter: `EFT.q` is the mathematical object, `s`q` is the program variable.

Types

A more substantial change has to do with the types. As stated before in section 2.3.2, this is mainly because of the TCCs generated by PVS. I implemented both solutions (changing types and adding context to the definitions):

n, k New type is $\mathbb{N}^+$, not $\mathbb{N}$. $n < 1$ is trivial and not really interesting. The type $\mathbb{N}^+$ has the intrinsic property > 0, which makes specification simpler, because $[0..n)$ is never empty.

h New type is $\mathbb{Z} \rightarrow \mathbb{N}$, not $[0..n) \rightarrow \mathbb{R}$. The return type is equal to that of the mathematical definition, which I discussed in section 4.1.1. The argument type has been broadened from $\mathbb{N}$ to $\mathbb{Z}$, because of problematic TCCs that arose from the former implementation.

w New type is $\mathbb{Z}$, not $\mathbb{N}$. This is simply due to some problematic TCCs.

t', at' New type is $\mathbb{Z} \rightarrow \mathbb{N}$, not $[0..n) \rightarrow \mathbb{Z}$. Analogously to `h`, $[0..n)$ has been replaced by $\mathbb{N}$, which in its turn has been replaced by $\mathbb{Z}$ because of problematic TCCs. The return type has been narrowed down to $\mathbb{N}$, because these variables represent indexes of data elements, which are in the range $[0..n]$, so there is no need to include negative numbers.

FT' New type is $\mathbb{N} \rightarrow \mathbb{P}(\mathbb{N})$, not $[0..n) \rightarrow \mathbb{P}(\mathbb{Z})$. $[0..n)$ has been replaced by $\mathbb{N}$ and the return type has been narrowed to $\mathbb{N}$, following the same argumentation as t' and at'.

t'(q') $\leftarrow$ (w $\geq$ 0 ? w : 0) (Update, line 4). Here I added the context to the statement. Line 1 assigns a value to w. State condition $P0_u$ states that $u(k, h, n) = min(n, w)$, while the if-guard (line 2) states that $w < n$, thus $u(k, h, n) = w$. Now $u(k, h, n)$ is of type $\mathbb{N}$, so $w \geq 0$. Therefore, the added context always reduces to true and the statements have the same functionality.

y1 $\leftarrow$ (t'(j + 1) - 1 $\geq$ 0 ? t'(j + 1) - 1 : 0) (FTHarvest, line 5). Here I added the context again. Lemma `t_positive` verifies that the function-value of t is > 0 if j + 1 > 0, which is true because the type of j is $\mathbb{N}$. The while-guard guarantees that $j \leq q$, so that property `stacks_filled` (part of the invariant) guarantees that t'(j + 1) = t(j + 1, n, h, n) and the program variable corresponds with the mathematical function. Again, the added context reduces to true and the statement has the equal effect as the original statement.

Altering the harvest phase

There is one last alteration in the algorithm, which is in the harvest phase. The initialisation of the inner for-loops has been moved to outside the outer loop (of FTHarvest). The reason for doing so is that I could specify a variant function that decreases in both outer- and inner loops. Of course, this change does not change any functionality, so I argue that the algorithm still calculates the same.

Because the only thing that has changed is the initialisation, it suffices to prove that the initialisation is correct. The first outer loop is easy: j has value 0, so that y $\leftarrow$ t'(j) reduces to y $\leftarrow$ t'(0), so y gets value 0. r $\leftarrow$ at'(j) + 1 reduces to r $\leftarrow$ at'(0) + 1, so that initialisation is correct as well.

When the SimpleHarvest-loop (lines 27-30) exits, y has the value y1 + 1, which reduces to t(j + 1) - 1 + 1 = t(j + 1). At the end of the FTHarvest-loop, j is incremented, so y = t(j). Another initialisation before the next SimpleHarvest- loop is not required. The argumentation for the MultiHarvest-loop (lines 31-38) runs analogously: after the inner MultiHarvest-loop, r has the value at(j + 1) + 1. Incrementing j means that r has the value at(j) + 1, so initialisation before the next inner MultiHarvest- loop is superfluous.

4.2 Proof

In this section I explain the mechanical verification of the EFT itself. I do not repeat every detail of which the proof consists, but I try to sketch the thought process behind the proof.

In section 4.2.1 I do a bottom-up analysis the proof, by building up the mathematics required for the algorithm. This roughly corresponds with the file EFT.pvs. Section 4.2.2 contains a top-down analysis, placing the mathematics in the right context. This analysis is formalised in the file EFT_program.pvs.

4.2.1 Mathematics

The problem with the definition of EFT is that direct calculation is computationally expensive. Calculating the *edt* is done by comparing each pair of data points. Then to find the EFT, at each point the distance is compared to the *edt*. With tricks like dynamic programming, the complexity could be reduced, but it still requires comparing each pair of data points, so the result is at least $\mathcal{O}(n \log n)$, which is not good enough.

Mathematical definitions

The first step is to introduce a new upper bound k for the range of p in definition (3.5), so the new range op p is $[0..k)$ while the range of x remains $[0..n)$. Now let $M(x, k, h)$ be the set of "background" points in the range $[0..k)$ which have a smaller distance to x than all other "background" points in the same range:

$$M(x, k, h) = \{p \mid p < k \wedge \forall q : q < k \Rightarrow f(x, p, h) \leq f(x, q, h)\} \qquad \text{(definition M)}$$

For $k = n$, M covers the full range, so $M(x, n, h) = EFT(x, n, h)$. According to (definition M): $M(x, 1, h) = 0$, because the range $[0..1)$ contains only 0. We search for an inductive definition of M in order to compute M. Incrementing k adds one possible value to the range: $p = k$, so comparing this to an element already in M gives the following equation:

$$p \in M(x, k, h) \Rightarrow \qquad \text{(M_inductive)}$$
$$M(x, k+1, h) = \begin{cases} M(x, k, h) & \text{if } f(x, p, h) < f(x, k, h) \\ M(x, k, h) \cup \{k\} & \text{if } f(x, p, h) = f(x, k, h) \\ \{k\} & \text{otherwise} \end{cases}$$

If (`M_inductive`) is applied directly, $M(x, n, h)$ can be calculated in n steps. But M needs to be calculated for each x, resulting in an $\mathcal{O}(n^2)$ algorithm. To solve that problem, the (non-decreasing) monotonicity of M is used:

$$x < y \wedge p \in M(x, k, h) \wedge q \in M(y, k, h) \Rightarrow \qquad \text{(M_nondecreasing)}$$
$$p \leq q$$

This is true, because when $p \in M(x, k, h)$ and $q \in M(y, k, h)$, it follows that $(q - p)(y - x) \geq 0$.

Let $a(x, k, h, n)$ be the minimum element of $M(x, k, h)$, which always exists (verified by `a_TCC2`):

$$a(x, k, h, n) = \begin{cases} k - 1 & \text{if } x = n \\ min(M(x, k, h)) & \text{otherwise} \end{cases} \qquad \text{(definition a)}$$

The case $x = n$ is outside the data range, but choosing the value $k - 1$ for this case turns out quite convenient later in (definition `u`). Because (`M_nondecreasing`), a is non-decreasing as well. Using this property, it is possible to further limit the range of p in the definition of M:

$$x < n \wedge k \leq n \Rightarrow \qquad \text{(M_def_a)}$$
$$M(x, k, h) = \{p \mid a(x, k, h, n) \leq p \wedge p \leq a(x+1, k, h, n) \wedge f(x, p, h) = f(x, a(x, k, h, n), h)\}$$

Assuming $a(x,n,h,n)$ can be calculated for all $x \in [0..n]$, the value of $M(x,n,h)$ $(= EFT(x,n,h))$ can be derived, by using an inductive formula for (`M_def_a`). A new upper bound r is introduced for p:

$$N(r,x,k,h,n) = \{p \mid a(x,k,h,n) \le p \wedge p < r \wedge f(x,p,h) = f(x,a(x,k,h,n),h)\} \qquad \text{(definition N)}$$

Replacing r with $a(x+1,k,h,n)$ in (definition `N`) returns the (`M_def_a`) again. Beginning at $r = a(x,k,h,n)$, the formula is rewritten to an inductive formula:

$$x < n \wedge k \le n \Rightarrow \qquad \text{(N_inductive)}$$
$$N(r+1,x,k,h,n) = \begin{cases} N(r,x,k,h,n) \cup \{r\} & \text{if } f(x,r,h) = f(x,a(x,k,h,n),h) \\ N(r,x,k,h,n) & \text{otherwise} \end{cases}$$

This concludes the mathematics for the harvest phase. What remains is to compute $a(x,n,h,n)$ for all $x \in [0..n]$. Therefore, the remainder of this section aims to find a formula for $a(x,k+1,h,n)$ in terms of $a(x,k,h,n)$. Remember that $M(x,1,h)$ contains only one element, 0, so $a(x,1,h,n) = 0$ as well, so we start from $k = 1$ and then increment k.

It follows from the monotonicity and the range of M that there is one final segment in the range $[0..n]$, for which $a(x,k+1,h,n) = k$. Note that this segment is never empty, because of the case $x = n$ in the definition of a. Let $u(k,h,n)$ be the first data point in this segment:

$$u(k,h,n) = min\{x \mid a(x,k+1,h,n) = k\} \qquad \text{(definition u)}$$

Because of (`M_nondecreasing`), (definition `u`) and (definition `M`), replacing p with $a(x,k+1,h,n)$ and q with k, it follows that:

$$x < n \Rightarrow \qquad \text{(u_iff_f)}$$
$$(x < u(k,h,n) \equiv f(x,a(x,k,h,n),h) \le f(x,k,h))$$

Applying (`u_iff_f`) to (`M_inductive`), replacing p with $a(x,k,h,n)$, gives the inductive formula for a:

$$x < n \Rightarrow \qquad \text{(a_inductive)}$$
$$a(x,k+1,h,n) = \begin{cases} a(x,k,h,n) & \text{if } x < u(k,h,n) \\ k & \text{otherwise} \end{cases}$$

In order to compute the value of $u(k,h,n)$, the function g is introduced, which can be applied to (`u_iff_f`):

$$g(p,q,h) = \begin{cases} \left\lfloor \frac{q*q-p*p+h(q)-h(p))}{2*(q-p)} \right\rfloor & \text{if } p < q \\ 0 & \text{otherwise} \end{cases} \qquad \text{(definition g)}$$

The case $p < q$ should always be true, it is only added context to be able to prove `g_TCC1`. Therefore, the case is added to the precondition of the following formula.

$$p < q \Rightarrow \qquad \text{(f_iff_g)}$$
$$(f(x,p,h) \le f(x,q,h) \equiv x \le g(p,q,h))$$

Actually, g is derived from solving the inequation (`f_iff_g`) for x. Since g can be computed directly, combining (`f_iff_g`) and (`u_iff_f`) gives the value of u (the exact values that are used for g are discussed later with lemma (`u_eq_g`)).

The function a is non-decreasing, but we can represent it by an increasing set, leaving out all the points where a remains constant (how to reconstruct all values of a is discussed later with lemma (`a_eq_at`)):

$$Q(k,h,n) = \{x \mid x = 0 \vee 1 \le x \wedge x < n \wedge a(x-1,k,h,n) < a(x,k,h,n)\} \qquad \text{(definition Q)}$$

Again an inductive formula is required, to compute $Q(k+1,h,n)$ from $Q(k,h,n)$. Incrementing k to $k+1$ means that k is added as possibly nearest background points. Because of the monotonicity of a in x, this point affects only the last segment, which is marked by u, see (definition `u`). Before this point the data remains unaffected, by (`a_inductive`).

$$Q(k+1,h,n) = \{x \mid x < u(k,h,n) \wedge x \in Q(k,h,n) \vee x < n \wedge x = u(k,h,n)\} \qquad \text{(Q_inductive)}$$

Data representation

The set Q minimizes the required amount of data to be stored, but how do you actually store it? The answer is to use a stack that enumerates the elements of Q in an increasing order. Unfortunately, this seemingly simple notion of enumerating set elements translates poorly to PVS.

In order to get the top of the stack, the elements in Q need to be counted:

$$q(k,h,n) = \#Q(k,h,n) \qquad \text{(definition q)}$$

The function t enumerates the elements of Q:

$$t(i,k,h,n) = ith_element(i, Q(k,h,n), n) \qquad \text{(definition t)}$$

In order to understand that definition, the implementation of `ith_element` is required:

$$ith_element(i,U,upb) = \begin{cases} min(U \cup \{upb\}) & \text{if } i = 0 \\ ith_element(i-1, U \setminus \{min(U \cup \{upb\})\}, upb) & \text{otherwise} \end{cases}$$
(definition `ith_element`)

What happens in (definition `ith_element`) is that in each recursive step the smallest element is removed from U. The problem is that U could be empty and there is no smallest element. Therefore an element upb is added to U before the minimum is taken, upb standing for upper bound. For the definition to work as intended, it is required that $\forall x : x \in U \Rightarrow x < upb$. In (definition `t`), it turns out that choosing n fulfils this requirement.[2]

(definition `ith_element`) is a definition that not only looks imposing, but is also difficult to build proofs with. Therefore, the file ith_element_theory.pvs not only gives the definition, but also states many useful properties about the definition that appear simple enough, but are not always simple to prove, such as the monotonicity of `ith_element`.

Not only are the index values of the set Q required, but also the value of a at these points. This is specified in at:

$$at(i,k,h,n) = a(t(i,k,h,n),k,h,n) \qquad \text{(definition at)}$$

This function helps to restore the value of $a(x,k,h,n)$ if $x \notin Q(k,h,n)$:

$$i < q(k,h,n) \wedge t(i,k,h,n) \le x \wedge x < t(i+1,k,h,n) \Rightarrow a(x,k,h,n) = at(i,k,h,n) \qquad \text{(a_eq_at)}$$

[2] It turns out that n is the only valid choice for upb, but this is not discussed until section 4.2.2.

It also provides a value that can be applied to (`u_iff_f`):

$$i < q(k,h,n) \Rightarrow \qquad (\texttt{tu_iff_f})$$
$$(t(i,k,h,n) < u(k,h,n) \equiv f(t(i,k,h,n), at(i,k,h,n), h) \leq f(t(i,k,h,n), k, h))$$

By (`Q_inductive`), all $t(i,k,h,n) \geq u(k,h,n)$ are not in the set $Q(k+1,h,n)$, and the one before the biggest i with that property remain in the set. Therefore the biggest i needs to be found. This is specified in $maxI$:[3]

$$maxI(k,h,n) = \begin{cases} max\{i \mid i < q(k,h,n) \wedge t(i,k,h,n) < u(k,h,n)\} & \text{if } 0 < u(k,h,n) \\ 0 & \text{otherwise} \end{cases} \qquad (\text{definition } \texttt{maxI})$$

This value can be found by linear search from high to low, searching for a value of i where $f(t(i,k,h,n), at(i,k,h,n), h) \leq f(t(i,k,h,n), k, h)$, using (`u_iff_f`).

It follows from (definition `u`) and (definition `Q`) that if $u(k,h,n) = 0$, then Q contains only 0. This case is simple and needs no further analysis. The other case where u is positive is more interesting. Two cases can be distinguished: $u(k,h,n) < n$ and $u(k,h,n) = n$. By (definition `u`), the value can not be higher than n. Using (`u_iff_f`) and (`f_iff_g`), we can compute u by computing g, only this value can be $> n$. The following definition catches that case:

$$0 < u(k,h,n) \Rightarrow \qquad (\texttt{u_eq_g})$$
$$u(k,h,n) = min(n, 1 + g(at(maxI(k,h,n), k, h, n), k, h))$$

What remains is to find the inductive definitions of q, t and at. (`Q_inductive`) adds element $u(k,h,n)$ to $Q(k+1,h,n)$ when $u(k,h,n) < n$, while in the case $u(k,h,n) = n$ this element is left out. From (definition `maxI`) and monotonicity of t follows $\forall i : i \leq maxI(k,h,n) \Rightarrow t(i,k,h,n) < u(k,h,n)$, thus these elements are all in $Q(k+1,h,n)$, according to (`Q_inductive`). This leads to the inductive definitions:

$$0 < u(k,h,n) \Rightarrow \qquad (\texttt{q_inductive})$$
$$q(k+1,h,n) - 1 = maxI(k,h,n) + (u(k,h,n) < n \ ? \ 1 : 0)$$

$$0 < u(k,h,n) \wedge i \leq maxI(k,h,n) \Rightarrow \qquad (\texttt{t_inductive_bounded})$$
$$t(i,k+1,h,n) = t(i,k,h,n)$$

$$0 < u(k,h,n) \wedge u(k,h,n) < n \Rightarrow \qquad (\texttt{t_inductive_newElement})$$
$$t(maxI(k,h,n) + 1, k+1, h, n) = u(k,h,n)$$

$$0 < u(k,h,n) \wedge i \leq maxI(k,h,n) \Rightarrow \qquad (\texttt{at_inductive_bounded})$$
$$at(i,k+1,h,n) = at(i,k,h,n)$$

This concludes the algorithmic analysis. The following section will show how the above mathematical notions are used in the algorithm.

[3] Although PVS provides no definition for max, the auxiliary file max_nat.pvs contains an definition that works very intuitive.

4.2.2 Algorithm

The algorithm is stated in this section, using the standard notation for annotated algorithms with Hoare triples. The standard strategies for proving loops and statements apply to most of the algorithm, but not to all. Only the non-standard parts of the proof are discussed in this section. I will give the full preconditions, postconditions and invariants for each part of the algorithm. Corresponding to the way the PVS file is built up, I will build the algorithm from the bottom up, creating the complete loop body before the encompassing loop.

Recall that the program variables have primes, whereas their mathematical counterparts are unprimed.

Build phase

The build phase consists of an outer while-loop, incrementing k. The inductive definitions that increment k apply to this section. The goal of the build phase is to build up the stacks t' and at', so that eventually they contain the values of (definition `t`) and (definition `at`) for $k = n$.

To avoid repetition in the predicates, to keep a clear overview and to ease the proofs a bit (using the `mixedHoare` lemma), the predicates have been split up into shorter ones. Throughout the build phase, the following (shorter) predicates occur on multiple places and have been split up:

$$t'(0) = 0 \tag{T_0}$$

$$T_0 \wedge k < n \tag{I_b}$$

`stacks_partially_filled` (Spf) states that the stacks t' and at' are filled up to $q(k,h,n)$, with the corresponding function value:

$$q' = q(k,h,n) - 1 \wedge (\forall i : i \leq q' \Rightarrow t'(i) = t(i,k,h,n) \wedge at'(i) = at(i,k,h,n)) \tag{Spf}$$

Linear Search

The linear search fragment searches for the value of maxI. Remember that it is in the body of the outer build loop, so k remains fixed.

$$\begin{aligned} & q' = -1 \wedge u(k,h,n) = 0 \vee \\ & q' \geq 0 \wedge q' = maxI(k,h,n) \wedge 0 < u(k,h,n) \wedge \\ & (\forall i : i \leq q' \Rightarrow t'(i) = t(i,k+1,h,n) \wedge at'(i) = at(i,k+1,h,n)) \wedge \\ & at'(q') = at(q',k,h,n) \end{aligned} \tag{Q_{ls}}$$

Note that $\wedge$ has a higher precedence than $\vee$, so Q_{ls} is a disjunction of conjunctions.

$$\begin{aligned} & -1 \leq q' \wedge q' \leq q(k,h,n) - 1 \wedge \\ & (q' < q(k,h,n) - 1 \Rightarrow f(t(q'+1,k,h,n), at(q'+1,k,h,n), h) > f(t(q'+1,k,h,n),k,h)) \wedge \\ & (\forall i : i \leq q' \Rightarrow t'(i) = t(i,k,h,n) \wedge at'(i) = at(i,k,h,n)) \end{aligned} \tag{J_{ls}}$$

Algorithm 3 LinearSearch: Find $maxI(k, h, n)$

Precondition: Pre_{ls}: $I_b \wedge Spf$
Postcondition: $Post_{ls}$: $I_b \wedge Q_{ls}$

$\{ \ Pre_{ls} \ \}$ (∗ `inv_init_ls` ∗)
$\{ \ Inv_{ls}$: $I_b \wedge J_{ls} \ \}$
(∗ $vf_{ls} = vf_b = 2(n - k) + q'$ ∗)

1. **while** q' ≥ 0 ∧ f(t'(q'), at'(q'), h) > f(t'(q'), k, h) **do** (∗ B_{ls} ∗)
   $\{ \ Inv_{ls} \wedge B_{ls} \ \}$
2. q' ← q' - 1
   $\{ \ Inv_{ls} \ \}$
3. **end while**

$\{ \ Inv_{ls} \wedge \neg B_{ls} \ \}$
(∗ `post_implied_ls`: $I_b \wedge J_{ls} \wedge \neg B_{ls} \Rightarrow I_b \wedge Q_{ls}$ ∗)
$\{ \ Post_{ls} \ \}$

There are many things going on here, especially in the definitions of Q_{ls} and J_{ls}. Notice that B_{ls} is a conjunction of two conditionals, that correspond to the disjunction Q_{ls}. If the loop is exited on the first condition $q' < 0$, the first disjunct of Q_{ls} becomes true. The difficult part is in the second disjunct, where J_{ls} and $\neg B_{ls}$ must imply Q_{ls}. The important conjunct to prove is $q' = maxI(k, h, n)$, the rest is not too hard, using the inductive definitions of section 4.2.1.

The trick is in the second line of J_{ls}, which contains an extra conditional $q' < q(k, h, n) - 1$, that becomes true once the loop has been entered once. If the loop has not been entered, $q' = maxI(k, h, n)$, because it is the biggest element in the range of i, see (definition `u`). If the loop has been entered at least once, $q' < q(k, h, n) - 1$ becomes true. The second line of J_{ls} combined with $\neg B_{ls}$ and (`tu_iff_f`) proves that $q' = maxI(k, h, n)$.

Reset

The reset fragment, also in the build loop, comes after the linear search fragment and is entered when the first disjunct of Q_{ls} is true. The goal is to reset the stacks to contain one correct element again.

$$T_0 \wedge u(k, h, n) = 0 \qquad (P_r)$$

Algorithm 4 Reset: Reset the stacks when $u(k) = 0$

Precondition: Pre_r: $I_b \wedge P_r$
Postcondition: $Post_r$: $I_b \wedge Spf[k + 1/k]$

$\{ \ Pre_r \ \}$

1. q' ← 0; at'(0) ← k
   $\{ \ Post_r \ \}$

This justifies the introduction of predicate T_0. It makes sure that $t'(0)$ does not have to be reset to 0 every time.

Update

The update fragment, also in the build loop, comes after the linear search fragment and is entered when the second disjunct of Q_{ls} is true. The goal is to update the stacks with the new element (if this applies).

$$\begin{aligned} & q' = maxI(k, h, n) \wedge 0 < u(k, h, n) \ \wedge \\ & (\forall i : i \leq q' \Rightarrow t'(i) = t(i, k+1, h, n) \wedge a't(i) = at(i, k+1, h, n)) \ \wedge \\ & a't(q') = at(q', k, h, n) \end{aligned} \qquad (P_u)$$

Algorithm 5 Update: Update the stacks when $0 < u(k)$

Precondition: Pre_u: $I_b \wedge P_u$
Postcondition: $Post_u$: $I_b \wedge Spf[k+1/k]$

```
        { Pre_u }
           (* u_eq_g ∧ q' = maxI(k,h,n) ∧ at'(q') = at(q',k,h,n) *)
        { Pre_u ∧ u(k,h,n) = min(n, 1 + g(at'(q'),k,h)) }
 1. w ← 1 + g(at'(q'), k, h)
        { P_u ∧ u(k,h,n) = min(n,w) }
 2. if w < n then
            { Pre_u ∧ 0 ≤ u(k,h,n) = w < n }
               (* (q_inductive) ∧ (t_inductive_newElement) *)
            { Post_u[q'+1/q', w/t'(q'+1), k/at'(q'+1)] ∧ w ≥ 0 }
 3.         q' ← q' + 1
            { Post_u[w/t'(q'), k/at'(q')] ∧ w ≥ 0 }
 4.         t'(q') ← ( w ≥ 0 ? w : 0 ); at'(q') ← k
            { Post_u }
 5. else
            { Pre_u ∧ u(k,h,n) = n }
               (* (q_inductive) ∧ (t_inductive_newElement) *)
            { Post_u }
 6.         SKIP
            { Post_u }
 7. end if
        { Post_u }
```

The if-else statement distinguishes between the case $u(k,h,n) < n$ and $u(k,h,n) = n$, as required by the inductive definitions of q and t.

In the above proof I have added $w \geq 0$, as required for line 4. It follows from line 1 and 2 that this condition is true, however, this is **not** mechanically verified.

Build

What remains is to tie the above fragments together. This is done in the build fragment, completing the build phase. The precondition `fullset[state]` applies here, leading to the following predicates:

The postcondition is equivalent to $Spf[n/k]$:

$$q' = q(n,h,n) - 1 \wedge (\forall i : i \leq q' \Rightarrow t'(i) = t(i,n,h,n) \wedge at'(i) = at(i,n,h,n)) \qquad (Q_b)$$

$$k \leq n \wedge Spf \qquad (J_b)$$

Algorithm 6 Build: Build the stacks t' and at'

Precondition: Pre_b: $\top$
Postcondition: $Post_b$: Q_b

```
        { ⊤ }
 1. q' ← 0; t'(0) ← 0; at'(0) ← 0
        { T_0 ∧ Spf[1/k] }
 2. k ← 1
        { T_0 ∧ Spf }    (* 1 ≤ n ∧ spf ⇒ J_b *)
        { Inv_b: T_0 ∧ J_b }
           (* vf_b = 2(n − k) + q' *)
 3. while k < n do      (* B_b *)
            { Inv_b ∧ B_b }
               (* T_0 ∧ J_b ∧ B_b ⇒ I_b ∧ Spf *)
            { Pre_ls: I_b ∧ Spf }
 4.      LinearSearch
            { Post_ls: I_b ∧ Q_ls }
 5.      if q' < 0 then
                 { Post_ls ∧ q' < 0 }
                    (* Select correct disjunct of Q_ls *)
                 { Pre_r: I_b ∧ P_r }
 6.           Reset
                 { Post_r: I_b ∧ Spf[k + 1/k] }
 7.      else
                 { Post_ls ∧ q' ≥ 0 }
                    (* Select correct disjunct of Q_ls *)
                 { Pre_u: I_b ∧ P_u }
 8.           Update
                 { Post_u: I_b ∧ Spf[k + 1/k] }
 9.      end if
            { I_b ∧ Spf[k + 1/k] }
10.      k ← k + 1
            { Inv_b }
11. end while
        { Inv_b ∧ ¬B_b }    (* k = n, post_implied_build *)
        { Post_b }
```

The build fragment simply initialises the values of q', t' and at' for $k = 1$, then updates k until $k = n$. The guard of "if" in line 5 guarantees that the correct disjunct of Q_{ls} is applied to the correct fragment, reset or update. I_b is made true after line 3 and remains true until just before line 10.

Harvest phase

The second phase of the algorithm is the harvest phase, the goal of which is to reconstruct the value of EFT from the built up stacks. Again, some predicates have been split up.

The reader might have noticed that the stacks are still said to be only *partially* filled. The reason for this name is that in order to harvest the stacks, an extra value is required. This is where the choice of n as upb in (definition `t`) becomes important.

$$q' = q(n,h,n) - 1 \wedge (\forall i : i \leq q' + 1 \Rightarrow t'(i) = t(i,n,h,n) \wedge at'(i) = at(i,n,h,n)) \qquad (Sf)$$

Note that $Sf \not\equiv Spf[n+1/k]$, it is just that one extra element has been added to the stacks. Because of Sf, the stacks values can be used as their corresponding mathematical function values. Again there is an invariant that remains true in the inner fragments.

$$Sf \wedge j \leq q' \wedge y1 = t'(j+1) - 1 \qquad (I_h)$$

Simple Harvest

The simple harvest fragment collects only the smallest element, a, in the variable EFT'. Collecting happens by using (definition `N`).

$$(\forall x : x < t'(j) \Rightarrow EFT'(x) = EFT(x, n, h)) \wedge \\ y = t'(j) \wedge r = at'(j) + 1 \tag{P_{sh}}$$

$$(\forall x : x < y1 \Rightarrow EFT'(x) = EFT(x, n, h)) \wedge EFT'(y1) = N(at'(j) + 1, y1, n, h, n) \wedge \\ y = t'(j + 1) \wedge r = at'(j) + 1 \tag{Q_{sh}}$$

$$(\forall x : x < t'(j) \Rightarrow EFT'(x) = EFT(x, n, h)) \wedge \\ t'(j) \leq y \wedge y \leq t'(j + 1) \wedge (\forall(x) : t'(j) \leq x \wedge x < y \Rightarrow EFT'(x) = N(at'(j) + 1, x, n, h, n)) \wedge \\ r = at'(j) + 1 \tag{J_{sh}}$$

Algorithm 7 SimpleHarvest: Get the minumum element of EFT

Precondition: Pre_{sh}: $I_h \wedge P_{sh}$
Postcondition: $Post_{sh}$: $I_h \wedge Q_{sh}$

$\{ Pre_{sh} \}$ (∗ `inv_init_sh` ∗)
$\{ Inv_{sh}$: $I_h \wedge J_{sh} \}$
(∗ $vf_{sh} = vf_{fth} = 3n - j - y - r$ ∗)

1. **while** y ≤ y1 **do** (∗ B_{sh} ∗)
   $\{ Inv_{sh} \wedge B_{sh} \}$
2. EFT'(y) ← {at'(j)}
   $\{ I_h \wedge J_{sh}[y + 1/y] \}$
3. y ← y + 1
   $\{ Inv_{sh} \}$
4. **end while**

$\{ Inv_{sh} \wedge \neg B_{sh} \}$ (∗ `post_implied_sh` ∗)
$\{ Post_{sh} \}$

P_{sh} tells that the EFT has been found for the range $[0..t'(j))$, plus some information about y and r. Q_{sh} has extended this range to $[0..y1)$, while only part of the EFT has been found for $x = y1$, according to (definition `N`).

This is reflected in J_{sh}. The first line tells that the range of P_{sh} remains unaltered. The second line is about the range of y in the loop and states that for each x in that left side of that range $EFT'(x) = N(at'(j) + 1, x, n, h, n)$. The third line states that the value of p remains constant.

The `post_implied_sh` proof is the tricky part: it contains the proof that $EFT(x, n, h) = N(at'(j) + 1, x, n, h, n)$ for $x \in [t'(j)..y1)$. This follows from (`a_eq_at`) and (`M_def_a`), because p can only have one value: $at'(j)$.

Multi Harvest

The multi harvest fragment finishes what the simple harvest fragment started. It adds the remaining elements to $EFT(y1, n, h)$, using (definition `N`).

$$(\forall x : x \leq y1 \Rightarrow EFT'(x) = EFT(x, n, h)) \wedge \\ y = t'(j + 1) \wedge r = at(j + 1) + 1 \tag{Q_{mh}}$$

$$(\forall x : x < y1 \Rightarrow EFT'(x) = EFT(x,n,h)) \wedge \\ at'(j) < r \wedge r \leq at'(j+1) + 1 \wedge EFT'(y1) = N(r, y1, n, h, n) \wedge \\ y = t'(j+1) \tag{J_{mh}}$$

Algorithm 8 MultiHarvest: Get the full set $EFT(y1, n, h)$

Precondition: Pre_{mh}: $I_h \wedge Q_{sh}$
Postcondition: $Post_{mh}$: $I_h \wedge Q_{mh}$

$\{ Pre_{mh} \}$ (∗ `inv_init_mh` ∗)
$\{ Inv_{mh}$: $I_h \wedge J_{mh} \}$
(∗ $vf_{mh} = vf_{fth} = 3n - j - y - r$ ∗)

1. **while** r ≤ at'(j + 1) **do** (∗ B_{mh} ∗)
   $\{ Inv_{mh} \wedge B_{mh} \}$
2. **if** f(y1, r, h) = f(y1, at'(j), h) **then**
   (∗ $f(y1, r, h) = f(y1, at'(j), h) \wedge$ (`N_inductive`) ∗)
   $\{ I_h \wedge J_{mh}[r+1/r,\ EFT'(y1) \cup \{r\}/EFT'(y1)] \}$
3. EFT'(y1) ← EFT'(y1) ∪ {r}
   $\{ I_h \wedge J_{mh}[r+1/r] \}$
4. **else**
   (∗ $f(y1, r, h) \neq f(y1, at'(j), h) \wedge$ (`N_inductive`) ∗)
   $\{ I_h \wedge J_{mh}[r+1/r] \}$
5. SKIP
   $\{ I_h \wedge J_{mh}[r+1/r] \}$
6. **end if**
   $\{ I_h \wedge J_{mh}[r+1/r] \}$
7. r ← r + 1
   $\{ Inv_{mh} \}$
8. **end while**

$\{ Inv_{mh} \wedge B_{mh} \}$ (∗ `post_implied_mh` ∗)
$\{ Post_{mh} \}$

Nothing surprising happens in this fragment. The value of r gets incremented in the range $[at'(j)..at'(j+1)]$ (J_{mh} line 2), so that (`N_inductive`) is computed. The if guard (line 2), guarantees that only elements that are really in N are added to EFT'.

FT Harvest

The harvest fragments are tied together in the FT harvest fragment.

$$\forall x : x < n \Rightarrow EFT'(x) = EFT(x,n,h) \tag{Q_{fth}}$$

$$j \leq q' + 1 \wedge (\forall x : x < t'(j) \Rightarrow EFT'(x) = EFT(x,n,h)) \wedge \\ y = t'(j) \wedge r = at'(j) + 1 \tag{J_{fth}}$$

Algorithm 9 FTHarvest: Harvest EFT using t' and at'

Precondition: Pre_{fth}: Q_b
Postcondition: $Post_{fth}$: Q_{fth}

```
        { Pre_fth }
 1. t'(q' + 1) ← n; at'(q' + 1) ← n - 1
        { Sf }
            (* 0 ≤ q' + 1, 0 = t'(0) *)
        { Sf ∧ J_fth[0/j, 0/y, at'(0) + 1/r] }
 2. y ← 0; r ← at'(0) + 1
        { Sf ∧ J_fth[0/j] }
 3. j ← 0
        { Inv_fth: Sf ∧ J_fth }
            (* vf_fth = 3n − j − y − r *)
 4. while j ≤ q' do      (* B_fth *)
            { Inv_fth ∧ B_fth }
                (* Sf ⇒ t'(j + 1) = t(j + 1, n, h, n) > 0 *)
            { Inv_fth ∧ B_fth ∧ t'(j + 1) − 1 ≥ 0 }
 5.         y1 ← ( t'(j + 1) - 1 ≥ 0 ? t'(j + 1) - 1 : 0 )
            { Pre_sh: I_h ∧ P_sh }
 6.         SimpleHarvest
            { Post_sh ≡ Pre_mh }
 7.         MultiHarvest
            { Post_mh ≡ Inv_fth[j + 1/j] }
 8.         j ← j + 1
            { Inv_fth }
 9. end while
        { Inv_fth ∧ ¬B_fth }     (* post_implied_fth *)
        { Post_fth }
```

In this fragment, first the stacks get updated with the new value, so that Q_b becomes Sf. Then the values of y, r and j get initialised. The loop increases j until $j = q' + 1$, so that at the end the entire stack has been evaluated. `post_implied_fth` confirms that this indeed covers the entire range $[0..n)$.

In the above proof I have added $t'(j + 1) - 1 \geq 0$, as required for line 5. It is immediate from the precondition that this condition is true, however, this is **not** mechanically verified.

OneFT

One last step is required: tying the two phases together, which is done in the OneFT fragment.

Algorithm 10 OneFT: Get EFT in one dimension

Precondition: Pre_{oft}: $\top$
Postcondition: $Post_{oft}$: Q_{fth}

```
 1. procedure OneFT(n : ℕ⁺; h : ℤ → ℕ)
 2.         var q', w : ℤ; t', at' : ℤ → ℕ; k : ℕ⁺; j, y, y1, r : ℕ; EFT' : ℕ → ℙ(ℕ)
            { Pre_oft ≡ Pre_b ≡ ⊤ }
 3.         Build
            { Post_b ≡ Pre_fth }
 4.         FTHarvest
            { Post_oft ≡ Post_fth ≡ ∀x : x < n ⇒ EFT'(x) = EFT(x, n, h) }
 5.         return EFT'
 6. end procedure
```

4.2.3 Time complexity

In section 2.1.2 I explained how to derive the time complexity of algorithms, especially the ones with nested loops. The above algorithm fragments contain a variant function, denoted by vf just above the while-loop.

Indeed the vf of the entire build phase is the same: $vf_b = 2(n - k) + q'$, just as the vf of the harvest phase: $vf_{fth} = 3n - j - y - r$. If we enter the initial values of the variables in these functions, the order of time complexity can be derived. Initially, $vf_b = 2(n-1) + 0 = \mathcal{O}(n)$ and $vf_{fth} = 3n - 0 - 0 - at'(0) + 1$. Following from (Sf), at' corresponds with the function value at, which is equal to a according to (`a_eq_at`) and that is bounded by n according to `a_bounded`. Therefore $vf_{fth} = \mathcal{O}(n)$ and the entire algorithm is $\mathcal{O}(n)$.

The correctness of the variant functions has been mechanically verified. What has not been mechanically verified is the conclusion about the time complexity.

Chapter 5

Conclusion and evaluation

The correctness and time complexity of the specified Euclidean Feature Transform algorithm in one dimension have been mechanically verified successfully, see chapter 4. The specified function EFT is computed in $\mathcal{O}(n)$ by the specified algorithm. A total of 141 PVS theorems were required to prove the correctness of the algorithm, distributed over 8 PVS specification files.

The assumptions underlying this project have been given in chapter 2, which shows that very plausible assumptions were made in this project and that he number of assumptions is minimal. The assumptions contain two arguably weak points: the correctness of PVS and the correctness of the method to derive the time complexity from the variant function. Despite these points, the conclusion from chapter 2 is that the discussed mechanical verification is indeed a formal and correct mathematical proof.

Chapter 3 proves that the specified EFT function is correct. It also proves that this one dimensional solution provides (a part of) the solution in higher dimensions.

Evaluation

The EFT algorithm is correct, but this is not something I expected to disprove in this project. I understood the provided proof with the little mathematical knowledge I posses, therefore I believe intuition remains a very powerful "tool" for proving algorithms and mathematics. I did not encounter any deviations from the provided proof, it only lacked details that were required for mechanical verification.

What misled me, is the amount of work that is required to mechanically verify the EFT algorithm. The original algorithm consists of only 31 lines of code, which is practically nothing compared to modern software implementations, yet it took more than six months to verify its correctness. That does not even include the higher dimension algorithm.

It must be said that I did not use the automation provided by PVS in a lot of places. I tried the `grind` command at several points, but was disappointed at how little it helped in most situations. My personal belief is that mechanical verification will only become a useful tool once the level of automation will increase significantly.

Chapter 6

Future work

There are many aspects of the project that have been proven, but have not been mechanically verified. In order to minimize the amount of assumptions and possible errors, many of these proofs can be verified as well. For each of these projects the most important factor to be considered is: does the estimated amount of work weigh up to the achieved result?

Variables h and n

As mentioned in section 4.1.1, the procedure parameters h and n should not have been declared as variables. In that section I also gave possible solutions to fix this: strengthen the precondition and postcondition, or change the declaration of h and n.

Reduction of dimensions

The most logical next step would be to mechanically verify the dimension reduction, in section 3.2.1. Assuming this can be done, the verification is probably only useful if it can be applied to an algorithm. An implementation in any dimension would suffice, but a more powerful verification would be one of an algorithm in arbitrary dimension. This means that such an algorithm must be found first.

On the other hand, this process can be reversed, by first creating the proof and subsequently deriving an algorithm for arbitrary dimension from the proof. This technique is the same as the one proposed in the Program Correctness course. In this case, proving the correctness of the reduction of dimensions is not only valuable from a research perspective, but also from a practical perspective.

Deriving time complexity from the variant function

The method from section 2.1.2, deriving the time complexity from the variant function, is one that is not mechanically verified. This would require an extension of the file programs.pvs, which could strengthen many other proofs about time complexity of algorithms.

Converting the algorithm

During the project, I spent quite some time investigating the built-in PVS data structures, hoping they would reduce the amount of work that went into *ith_element_theory.pvs*. But because PVS is implemented in lisp, the built-in data structures are functional ones. If a functional implementation of the algorithm could be constructed, this would make the PVS implementation of the proof easier. This could be especially lucrative when an implementation in arbitrary dimension is sought.

PVS

A totally different approach would be to extend PVS. More automation could be added so that proving becomes less laborious. But one could also try to verify the PVS implementation. Before doing so, a research project comparing other proof assistants is probably in order.

Chapter 7

Acknowledgement

I would like to thank Wim Hesselink for his assistance with PVS. His expertise with PVS and the many hours he has assisted me, have made the completion of this project possible. I have learned many techniques from him, most about PVS, but also more general proof techniques.

I would also like to thank Gerard Renardel for helping me with bringing together my over-abundant ambitions into a project that I was enthusiastic for, but that was fitted for a bachelor project.

Bibliography

[1] G. Borgefors, "Distance transformations in arbitrary dimensions," *Computer Vision, Graphics, and Image Processing*, vol. 27, no. 3, pp. 321 – 345, 1984. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0734189X84900355

[2] W. H. Hesselink, "A linear-time algorithm for euclidean feature transform sets," *Information Processing Letters*, vol. 102, no. 5, pp. 181 – 186, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019006003681

[3] W. H. Hesselink, "Distance transforms and feature transform sets," http://www.cs.rug.nl/~wim/imageproc/whh426.pdf, May 2009, an extension and modification of the IPL paper[2].

[4] W. H. Hesselink and J. B. T. M. Roerdink, "Euclidean skeletons of digital image and volume data in linear time by the integer medial axis transform," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, no. 12, pp. 2204 – 2217, dec. 2008.

[5] A. Meijster, J. B. T. M. Roerdink, and W. H. Hesselink, "A general algorithm for computing distance transforms in linear time," *Mathematical Morphology and its Applications to Image and Signal Processing*, vol. 27, pp. 321 – 345, 2000.

Appendix A

PVS Specification

Appendix A contains the complete PVS specification of the EFT algorithm and the proof of its correctness. Figure A.1 displays the dependencies for each of the proof files.

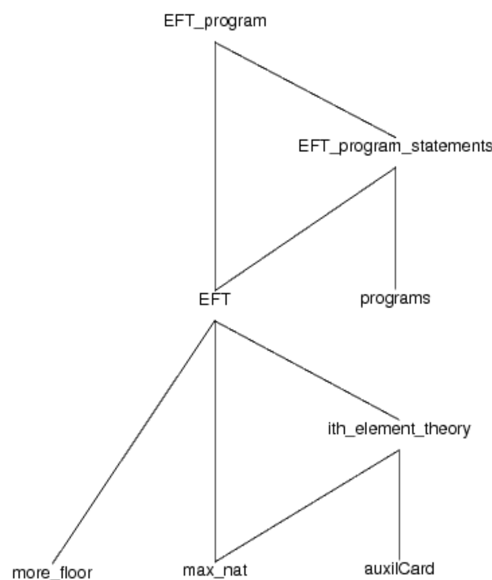

Figure A.1: PVS Proof hierarchy

A.1 EFT_program.pvs

The file EFT_program.pvs contains the proof of correctness for the EFT algorithm.

EFT_program.pvs

```
% The correctness of the EFT algorithm
%
% Author: Sebastian Verschoor
% Last modified: 28 July 2012
5
EFT_program: THEORY

BEGIN

10 IMPORTING EFT, EFT_program_statements


%%%%%%%%%%%%%%%%%%%%%
% PVS declacrations %
15 %%%%%%%%%%%%%%%%%%%%%
```

```
   % iterator over stack elements
   i: VAR nat

20 % iterator over data points
   x: VAR nat

   % program state
   s: VAR state
25

   %%%%%%%%%%%%%%%%%%%%
   % State predicates %
   %%%%%%%%%%%%%%%%%%%%
30
   % To avoid repetition in the predicates, to keep a clear overview and to ease to
   % proves a bit (using the mixedHoare-lemma), the predicates have been split up
   % into several shorter ones. The full predicates are conjunctions of the shorter
   % ones. For example, the full precondition of LinearSearch is "P_ls AND
35 % I_b".

   % Build phase. Parts of the build invariant are true in the entire while-body,
   % some just in part of the while-body.

40 % Build, after line 2 to before line 11: required for Reset (t_0)
   t_set_zero(s): bool =
     s`t(0) = 0

   % Build, after line 4 to before line 10
45 I_b(s): bool =
     t_set_zero(s) AND s`k < s`n

   % Build, part of loop-invariant: stacks filled up to q(k, h, n) (spf)
   stacks_partially_filled(s): bool =
50   s`q = EFT.q(s`k, s`h, s`n) - 1 AND
     ( FORALL(i): i <= s`q IMPLIES s`t(i)  = EFT.t(i, s`k, s`h, s`n) AND
                                   s`at(i) = EFT.at(i, s`k, s`h, s`n) )

   % Update/Reset, part of postcondition: stacks filled up to q(k+1, h, n)
55 % ( spf(k = k + 1) )
   % Postcondition of both Reset and Update
   stacks_partially_filled_next(s): bool =
     s`q = EFT.q(s`k + 1, s`h, s`n) - 1 AND
     ( FORALL(i): i <= s`q IMPLIES s`t(i)  = EFT.t(i, s`k + 1, s`h, s`n) AND
60                                 s`at(i) = EFT.at(i, s`k + 1, s`h, s`n) )


   % LinearSearch, precondition
   % Pre_ls: I_b AND P_ls
65 P_ls(s): bool =
     stacks_partially_filled(s)

   % LinearSearch, postcondition
   % Post_ls: I_b AND Q_ls
70 % Either the first line or the rest is true, depending on the exit-condition of
   % the loop. The first line is (part of) the precondition for Reset, the rest
   % for Update. The extra line about the value of at is required for Update.
   Q_ls(s): bool =
     s`q = -1 AND EFT.u(s`k, s`h, s`n) = 0 OR
75     s`q >= 0 AND s`q = EFT.maxI(s`k, s`h, s`n) AND 0 < EFT.u(s`k, s`h, s`n) AND
       ( FORALL(i): i <= s`q IMPLIES s`t(i) = EFT.t(i, s`k + 1, s`h, s`n) AND
                                     s`at(i) = EFT.at(i, s`k + 1, s`h, s`n) ) AND
       s`at(s`q) = EFT.at(s`q, s`k, s`h, s`n)

80 % LinearSearch, while-loop invariant
   % Inv_ls: I_b AND J_ls
   % To make the invariant strong enough for (J_ls AND NOT B_ls IMPLIES Q_ls), we
   % introduce a condition that is only valid when the loop has been entered at
   % least once (and s`q < EFT.q(k, h, n) - 1)
85 J_ls(s): bool =
     -1 <= s`q AND s`q <= EFT.q(s`k, s`h, s`n) - 1 AND
     ( s`q < EFT.q(s`k, s`h, s`n) - 1 IMPLIES
         EFT.f(EFT.t(s`q + 1, s`k, s`h, s`n), EFT.at(s`q + 1, s`k, s`h, s`n), s`h)
         > EFT.f(EFT.t(s`q + 1, s`k, s`h, s`n), s`k, s`h) )
90   AND
     ( FORALL(i): i <= s`q IMPLIES s`t(i)  = EFT.t(i, s`k, s`h, s`n) AND
                                   s`at(i) = EFT.at(i, s`k, s`h, s`n) )


95 % Reset, precondition
   % Pre_r: I_b AND P_r
   P_r(s): bool =
     t_set_zero(s) AND EFT.u(s`k, s`h, s`n) = 0
```

```
100 % Reset, postcondition
    % Post_r: I_b AND Q_r
    Q_r(s): bool =
      stacks_partially_filled_next(s)

105
    % Update, precondition
    % Pre_u: I_b AND P_u
    P_u(s): bool =
      s`q = EFT.maxI(s`k, s`h, s`n) AND 0 < EFT.u(s`k, s`h, s`n) AND
110   ( FORALL(i): i <= s`q IMPLIES s`t(i)  = EFT.t(i, s`k + 1, s`h, s`n) AND
                                    s`at(i) = EFT.at(i, s`k + 1, s`h, s`n) ) AND
      s`at(s`q) = EFT.at(s`q, s`k, s`h, s`n)

    % Update, postcondition
115 % Post_u: I_b AND Q_u
    Q_u(s): bool =
      stacks_partially_filled_next(s)

    % Update, after line 1: intermediate condition
120 P0_u(s): bool =
      P_u(s) AND EFT.u(s`k, s`h, s`n) = min(s`n, s`w)


    % Build, postcondition (spf(k = n))
125 % Post_b: Q_b
    Q_b(s): bool =
      s`q = EFT.q(s`n, s`h, s`n) - 1 AND
        ( FORALL(i): i <= s`q IMPLIES s`t(i)  = EFT.t(i, s`n, s`h, s`n) AND
                                      s`at(i) = EFT.at(i, s`n, s`h, s`n) )
130
    % Build, invariant
    % Inv_b: t_0 AND J_b
    J_b(s): bool =
      s`k <= s`n AND stacks_partially_filled(s)
135

    % Harvest phase. Again some predicates have been split off.

    % FTHarvest, part of invariant: stacks filled up to "q(n, h, n) + 1" (sf)
140  stacks_filled(s): bool =
      s`q = EFT.q(s`n, s`h, s`n) - 1 AND
      ( FORALL(i): i <= s`q + 1 IMPLIES s`t(i)  = EFT.t(i, s`n, s`h, s`n) AND
                                        s`at(i) = EFT.at(i, s`n, s`h, s`n) )

145 % FTHarvest, before line 6 to after line 7
    I_h(s): bool =
      stacks_filled(s) AND s`j <= s`q AND s`y1 = s`t(s`j + 1) - 1


150 % SimpleHarvest, precondition
    % Pre_sh: I_h AND P_sh
    P_sh(s): bool =
      ( FORALL(x): x < s`t(s`j) IMPLIES s`EFT(x) = EFT.EFT(x, s`n, s`h) ) AND
      s`y = s`t(s`j) AND s`r = s`at(s`j) + 1
155
    % SimpleHarvest, postcondition
    % Post_sh: I_h AND Q_sh
    Q_sh(s): bool =
      ( FORALL(x): x < s`y1 IMPLIES s`EFT(x) = EFT.EFT(x, s`n, s`h) ) AND
160   s`EFT(s`y1) = EFT.N(s`at(s`j) + 1, s`y1, s`n, s`h, s`n) AND
      s`y = s`t(s`j + 1) AND s`r = s`at(s`j) + 1

    % SimpleHarvest, while-loop invariant
    % Inv_sh: I_h AND J_sh
165 J_sh(s): bool =
      ( FORALL(x): x < s`t(s`j) IMPLIES s`EFT(x) = EFT.EFT(x, s`n, s`h) ) AND
      s`t(s`j) <= s`y AND s`y <= s`t(s`j + 1) AND
      ( FORALL(x): s`t(s`j) <= x AND x < s`y IMPLIES
                   s`EFT(x) = EFT.N(s`at(s`j) + 1, x, s`n, s`h, s`n) ) AND
170   s`r = s`at(s`j) + 1


    % MultiHarvest, precondition
    % Pre_mh: I_h AND P_mh
175 P_mh(s): bool =
      Q_sh(s)

    % MultiHarvest, postcondition
    % Post_mh: I_h AND Q_mh
180 Q_mh(s): bool =
      ( FORALL(x): x <= s`y1 IMPLIES s`EFT(x) = EFT.EFT(x, s`n, s`h) ) AND
      s`y = s`t(s`j + 1) AND s`r = s`at(s`j + 1) + 1
```

```
    % MultiHarvest, while-loop invariant
185 % Inv_mh: I_h AND J_mh
    J_mh(s): bool =
      ( FORALL(x): x < s`y1 IMPLIES s`EFT(x) = EFT.EFT(x, s`n, s`h) ) AND
      s`at(s`j) < s`r AND s`r <= s`at(s`j + 1) + 1 AND
      s`EFT(s`y1) = EFT.N(s`r, s`y1, s`n, s`h, s`n) AND
190   s`y = s`t(s`j + 1)


    % FTHarvest, precondition
    % Pre_fth: P_fth
195 P_fth(s): bool =
      Q_b(s)

    % FTHarvest, postcondition
    % Post_fth: Q_fth
200 Q_fth(s): bool =
      FORALL(x): x < s`n IMPLIES s`EFT(x) = EFT.EFT(x, s`n, s`h)

    % FTHarvest, invariant
    % Inv_fth: sf AND J_fth
205 J_fth(s): bool =
      s`j <= s`q + 1 AND
      ( FORALL(x): x < s`t(s`j) IMPLIES s`EFT(x) = EFT.EFT(x, s`n, s`h) ) AND
      s`y = s`t(s`j) AND s`r = s`at(s`j) + 1

210
    % OneFT, postcondition
    % Post_oft: Q_oft
    Q_oft(s): bool =
      Q_fth(s)
215

    %%%%%%%%%%%%%%%%%%%%
    % Variant functions %
    %%%%%%%%%%%%%%%%%%%%
220
    % Build phase, variant function
    vf_b(s): int =
      2 * (s`n - s`k) + s`q
    vf_ls(s): int = vf_b(s)
225
    % Harvest phase, variant function
    vf_fth(s): int =
      3 * s`n - s`j - s`y - s`r
    vf_sh(s): int = vf_fth(s)
230 vf_mh(s): int = vf_fth(s)


    %%%%%%%%%%%%%%%%%%%%
    % Prove correctness %
235 %%%%%%%%%%%%%%%%%%%%

    % LinearSearch, step 1
    post_implied_ls: LEMMA
      subset?(I_b AND J_ls AND NOT B_ls, I_b AND Q_ls)
240
    % LinearSearch, step 2
    % There is no active initialisation. This lemma is actually equal to:
    % tcHoare(I_b AND P_ls, lift(skip), I_b AND J_ls)
    inv_init_ls: LEMMA
245   subset?(I_b AND P_ls, I_b AND J_ls)

    % LinearSearch, step 3
    vf_isvariant_ls: LEMMA
      isVariant(vf_ls, I_b AND J_ls, B_ls, lift(body_ls))
250
    % LinearSearch, step 4
    inv_keptvalid_ls: LEMMA
      tcHoare(I_b AND J_ls AND B_ls, lift(body_ls), I_b AND J_ls)

255 % LinearSearch, step 5
    program_correct_ls: LEMMA
      tcHoare(I_b AND P_ls, program_ls, I_b AND Q_ls)


260 % Reset
    program_correct_reset: LEMMA
      tcHoare(I_b AND P_r, lift(command_reset), I_b AND Q_r)


265 % Update
```

```
    program_correct_update: LEMMA
      tcHoare(I_b AND P_u, lift(command_update), I_b AND Q_u)


270 % Build, step 1
    post_implied_build: LEMMA
      subset?(J_b AND NOT B_b, Q_b)

    % Build, step 2
275 % Since we make no assumptions about the state before the algorithm is
    % executed, the precondition is equal to true. The fullset[state] is the set
    % containing all states, so this is the "predicate" we use as precondition.
    inv_init_build: LEMMA
      tcHoare(fullset[state], lift(init_build), t_set_zero AND J_b)
280
    % Build, step 3
    vf_isvariant_build: LEMMA
      isVariant(vf_b, t_set_zero AND J_b, B_b, body_build)

285 % Build, step 4
    inv_keptvalid_build: LEMMA
      tcHoare(t_set_zero AND J_b AND B_b, body_build, t_set_zero AND J_b)

    % Build, step 5
290 loop_correct_build: LEMMA
      tcHoare(t_set_zero AND J_b, loop_build, Q_b)

    % Build
    program_correct_build: LEMMA
295   tcHoare(fullset[state], program_build, Q_b)


    % SimpleHarvest, step 1
    post_implied_sh: LEMMA
300   subset?(I_h AND J_sh AND NOT B_sh, I_h AND Q_sh)

    % SimpleHarvest, step 2
    % No active initialisation
    inv_init_sh: LEMMA
305   subset?(I_h AND P_sh, I_h AND J_sh)

    % SimpleHarvest, step 3
    vf_isvariant_sh: LEMMA
      isVariant(vf_sh, I_h AND J_sh, B_sh, lift(body_sh))
310
    % SimpleHarvest, step 4
    inv_keptvalid_sh: LEMMA
      tcHoare(I_h AND J_sh AND B_sh, lift(body_sh), I_h AND J_sh)

315 % SimpleHarvest, step 5
    program_correct_sh: LEMMA
      tcHoare(I_h AND P_sh, program_sh, I_h AND Q_sh)


320 % MultiHarvest, step 1
    post_implied_mh: LEMMA
      subset? (I_h AND J_mh AND NOT B_mh, I_h AND Q_mh)

    % MultiHarvest, step 2
325 % No active initialisation
    inv_init_mh: LEMMA
      subset? (I_h AND P_mh, I_h AND J_mh)

    % MultiHarvest, step 3
330 vf_isvariant_mh: LEMMA
      isVariant(vf_mh, I_h AND J_mh, B_mh, lift(body_mh))

    % MultiHarvest, step 4
    inv_keptvalid_mh: LEMMA
335   tcHoare(I_h AND J_mh AND B_mh, lift(body_mh), I_h AND J_mh)

    % MultiHarvest, step 5
    program_correct_mh: LEMMA
      tcHoare(I_h AND P_mh, program_mh, I_h AND Q_mh)
340

    % FTHarvest, step 1
    post_implied_fth: LEMMA
      subset? (stacks_filled AND J_fth AND NOT B_fth, Q_fth)
345
    % FTHarvest, step 2
    inv_init_fth: LEMMA
      tcHoare(P_fth, lift(init_fth), stacks_filled AND J_fth)
```

```
350 % FTHarvest, step 3
    vf_isvariant_fth: LEMMA
      isVariant(vf_fth, stacks_filled AND J_fth, B_fth, body_fth)

    % FTHarvest, step 4
355 inv_keptvalid_fth: LEMMA
      tcHoare(stacks_filled AND J_fth AND B_fth, body_fth,
              stacks_filled AND J_fth)

    % FTHarvest, step 5
360 loop_correct_fth: LEMMA
      tcHoare(stacks_filled AND J_fth, loop_fth, Q_fth)

    % FTHarvest
    program_correct_fth: LEMMA
365   tcHoare(P_fth, program_fth, Q_fth)


    % OneEFT
    % Precondition = true
370 program_correct_oneft: THEOREM
      tcHoare(fullset[state], program_oneft, Q_oft)

    END EFT_program
```

A.2 EFT_program_statements.pvs

The file EFT_program_statements.pvs contains the program statements in the PVS syntax.

EFT_program_statements.pvs

```
    % EFT algorithm statements
    %
    % Author: Sebastian Verschoor
    % Last modified: 26 July 2012
5
    EFT_program_statements: THEORY

    BEGIN

10  IMPORTING EFT, programs


    %%%%%%%%%%%%%%%%%%%%%%%%
    % Program declarations %
15  %%%%%%%%%%%%%%%%%%%%%%%%

    % Program state type, a list of variables and their types
    state: TYPE = [# q: int,                    % stack height
                     t, at: [int -> nat],       % stacks
20                   k: posnat,                 % build iterator
                     n: posnat,                 % data boundary
                     h: [nat -> nat],           % grey-value function
                     w: int,                    % separator function result
                     j: nat,                    % stack iterator
25                   y, y1: nat,                % harvest iterator (t)
                     r: nat,                    % upper bound of N
                     EFT: [nat -> setof[nat]]   % result storage
                   #]

30  % Program state
    s: VAR state


    %%%%%%%%%%%%%%%%%%%%%%
35  % Program statements %
    %%%%%%%%%%%%%%%%%%%%%%

    % The skip command does nothing and can be used for implementing the if (with
    % no else)
40  skip(s): state = s


    % LinearSearch, line 1: guard while-loop
    B_ls(s): bool =
45    s`q >= 0 AND EFT.f(s`t(s`q), s`at(s`q), s`h) > EFT.f(s`t(s`q), s`k, s`h)

    % LinearSearch, line 2: body while-loop
```

```
    body_ls(s): state =
      s WITH [`q := s`q - 1]
50
    % LinearSearch, lines 1-3: while-loop
    program_ls: program[state] =
      while(B_ls, lift(body_ls))

55
    % Reset, line 1
    command_reset(s): state =
      s WITH [`q := 0, `at(0) := s`k]

60
    % Update, line 1
    command1_update(s): state =
      s WITH [`w := 1 + EFT.g(s`at(s`q), s`k, s`h)]

65  % Update, line 2: if-guard
    ifguard_update(s): bool =
      s`w < s`n

    % Update, line 3: if-body
70  command2_update(s): state =
      s WITH [`q := s`q + 1]

    % Update, line 4: if-body
    % The IF-THEN-ELSE construction is introduced to avoid akward TCCs. In the
75  % program, s`w >= 0 is always true.
    command3_update(s): state =
      s WITH [`t(s`q) := IF s`w >= 0 THEN s`w ELSE 0 ENDIF, `at(s`q) := s`k]

    % Update, lines 3-4: composition if-body
80  command23_update: command[state] =
      command2_update ^ command3_update

    % Update, lines 2-7: if (no else)
    commandif_update: command[state] =
85    ifThenElse(ifguard_update, command23_update, skip)

    % Update, lines 1-7: composition
    command_update: command[state] =
      command1_update ^ commandif_update
90

    % Build, line 1: initialisation
    init1_build(s): state =
      s WITH [`q := 0, `t(0) := 0, `at(0) := 0]
95
    % Build, line 2: initialisation
    init2_build(s): state =
      s WITH [`k := 1]

100 % Build, lines 1-2: composition initialisation
    init_build: command[state] =
      init1_build ^ init2_build

    % Build, line 3: while-guard
105 B_b(s): bool =
      s`k < s`n

    % Build, line 5: if-guard
    ifguard_build(s): bool =
110   s`q < 0

    % Build, line 10: while-body
    body1_build(s): state =
      s WITH [`k := s`k + 1]
115
    % Build, lines 5-9: if-else
    command_ifelse_build: command[state] =
      ifThenElse(ifguard_build, command_reset, command_update)

120 % Build, lines 4-10: composition while-body
    body_build: program[state] =
      program_ls ^ lift(command_ifelse_build ^ body1_build)

    % Build, lines 3-11: while-loop
125 loop_build: program[state] =
      while(B_b, body_build)

    % Build, lines 1-11: composition
    program_build: program[state] =
130   lift(init_build) ^ loop_build
```

```
    % SimpleHarvest, line 1: while-guard
    B_sh(s): bool =
135   s`y <=  s`y1

    % SimpleHarvest, line 2: while-body
    body1_sh(s): state =
      s WITH [`EFT(s`y) := singleton(s`at(s`j))]
140
    % SimpleHarvest, line 3: while-body
    body2_sh(s): state =
      s WITH [`y := s`y + 1]

145 % SimpleHarvest, lines 3-4: composition while-body
    body_sh: command[state] =
      body1_sh ^ body2_sh

    % SimpleHarvest, lines 1-4: while-loop
150 program_sh: program[state] =
      while(B_sh, lift(body_sh))


    % MultiHarvest, line 1: while-guard
155 B_mh(s): bool =
      s`r <= s`at(s`j + 1)

    % MultiHarvest, line 2: if-guard
    ifguard_mh(s): bool =
160     EFT.f(s`y1, s`r, s`h) = EFT.f(s`y1, s`at(s`j), s`h)

    % MultiHarvest, line 3: if-body
    commandif_mh(s): state = % line 5
      s WITH [`EFT(s`y1) := add(s`r, s`EFT(s`y1))]
165
    % MultiHarvest, lines 2-6: if (no else)
    bodyif_mh: command[state] =
      ifThenElse(ifguard_mh, commandif_mh, skip)

170 % MultiHarvest, line 7: while-body
    body1_mh(s): state =
      s WITH [`r := s`r + 1]

    % MultiHarvest, lines 2-7: composition while-body
175 body_mh: command[state] =
      bodyif_mh ^ body1_mh

    % MultiHarvest, lines 1-8: while-loop
    program_mh: program[state] =
180   while(B_mh, lift(body_mh))


    % EFTHarvest, line 1: initialisation
    init1_fth(s): state =
185   s WITH [`t(s`q + 1) := s`n, `at(s`q + 1) := s`n - 1]

    % EFTHarvest, line 2: initialisation
    init2_fth(s): state =
      s WITH [`y := 0, `r := s`at(0) + 1]
190
    % EFTHarvest, line 3: initialisation
    init3_fth(s): state =
      s WITH [`j := 0]

195 % EFTHarvest, lines 1-3: composition initialisation
    init_fth: command[state] =
      init1_fth ^ init2_fth ^ init3_fth

    % EFTHarvest, line 4: while-guard
200 B_fth(s): bool =
      s`j <= s`q

    % EFTHarvest, line 5: while-body
    % The IF-THEN-ELSE construction is introduced to avoid akward TCCs. In the
205 % program, s`t always corresponds to the function EFT.t, which is always
    % positive (verified by LEMMA t_positive).
    body1_fth(s): state =
      s WITH [`y1 := IF s`t(s`j + 1) - 1 >= 0 THEN s`t(s`j + 1) - 1 ELSE 0 ENDIF]

210 % EFTHarvest, line 8: while-body
    body2_fth(s): state =
      s WITH [`j := s`j + 1]
```

```
% EFTHarvest, lines 5-8: composition while-body
body_fth: program[state] =
  lift(body1_fth) ^ program_sh ^ program_mh ^ lift(body2_fth)

% EFTHarvest, lines 4-9: while-body
loop_fth: program[state] =
  while(B_fth, body_fth)

% EFTHarvest, lines 1-9: composition
program_fth: program[state] =
  lift(init_fth) ^ loop_fth


% OneEFT, lines 3-4: composition
program_oneft: program[state] =
  program_build ^ program_fth

END EFT_program_statements
```

A.3 EFT.pvs

The file EFT.pvs contains the specification of the mathematics that underlay the EFT algorithm.

EFT.pvs

```
% Mathematics behind the EFT algorithm
%
% The proof is based upon the article by Wim H. Hesselink, May 4, 2009:
%   Distance transforms and feature transform sets
%   src: http://www.cs.rug.nl/~wim/imageproc/whh426.pdf
% In the proof I refer to equations in this article, using the tag "whh426".
%
% Author: Sebastian Verschoor
% Last modified: 26 July 2012


EFT: THEORY

BEGIN

IMPORTING more_floor, ith_element_theory, max_nat


%%%%%%%%%%%%%%%%%%%%%%
% PVS declarations %
%%%%%%%%%%%%%%%%%%%%%%

% "Fixed" variables, provided as parameters to the function. Inside the EFT
% theory, these are considered to be constants. I declared them as variables,
% making it possible to import this theory into a theory of the higher dimension
% EFT, where both n and h vary.

% Boundary of data
n: VAR posnat

% Grey-value function
h: VAR [nat -> nat]


% Actual variables

% With the introduction of h, there is no real distinction between background
% points and foreground points. Still, it helps to think about them in these
% terms.

% "Background" points
p, q, r: VAR nat

% "Foreground" points
x, y: VAR nat

% Distance
d: VAR nat

% Data iterator
k: VAR posnat

% Stack indexes
i, j: VAR nat
```

```
55

%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the EFT %
%%%%%%%%%%%%%%%%%%%%%%%%%
60
% One-dimensional squared Euclidean distance
dist(x, p): nat =
  (x - p) * (x - p)

65 % Distance, including h
f(x, p, h): nat =
  dist(x, p) + h(p)

% Nearest distance to any point
70 edt(x, n, h): nat =
  min({ d | EXISTS p: d = f(x, p, h) AND p < n })

% All points with the nearest distance
EFT(x, n, h): setof[nat] =
75  { p | p < n AND f(x, p, h) = edt(x, n, h) }


%%%%%%%%%%%%%%%%%%%%%%%%%
% Auxiliary Mathematics %
80 %%%%%%%%%%%%%%%%%%%%%%%%%

% Alternative definition of EFT, introducing k as new bound for background points
% whh426 (0)
M(x, k, h): setof[nat] =
85  { p | p < k AND FORALL (q): q < k IMPLIES f(x, p, h) <= f(x, q, h)}

M_bounded: LEMMA
  M(x, k, h)(p) IMPLIES p < k

90 % Proving that the definitions of M and EFT are equal
M_equals_EFT: LEMMA
  M(x, n, h) = EFT(x, n, h)

% Inductive definition of M
95 % whh426 (1)
M_inductive: LEMMA
  M(x, k, h)(p) IMPLIES
    M(x, k + 1, h) = IF f(x, p, h) < f(x, k, h)
                     THEN M(x, k, h)
100                  ELSE IF f(x, p, h) = f(x, k, h)
                            THEN add(k, M(x, k, h))
                            ELSE singleton(k)
                          ENDIF
                   ENDIF
105
% M is monotonically non-decreasing in its first argument
% whh426 (2)
M_nondecreasing: LEMMA
  x < y AND M(x, k, h)(p) AND M(y, k, h)(q) IMPLIES p <= q
110

% Minimal element of M, including upper bound
% whh426 (3)
a(x, k, h, n): nat =
115  IF x = n THEN k - 1 ELSE min(M(x, k, h)) ENDIF

a_bounded: LEMMA
  a(x, k, h, n) < k

120 % Auxiliary lemma for a_nondecreasing
p_leq_a: LEMMA
  x < n AND M(x, k, h)(p) IMPLIES p <= a(x + 1, k, h, n)

a_nondecreasing: LEMMA
125  x < n IMPLIES a(x, k, h, n) <= a(x + 1, k, h, n)

a_nondecreasing_general: LEMMA
  x <= y AND y <= n IMPLIES a(x, k, h, n) <= a(y, k, h, n)

130 % M defined by a
% whh426 (4)
M_def_a: LEMMA
  x < n AND k <= n IMPLIES
    M(x, k, h) = { p | a(x, k, h, n) <= p AND p <= a(x + 1, k, h, n) AND
135                    f(x, p, h) = f(x, a(x, k, h, n), h) }

```

```
% New definition of M, adding the new upper bound r
N(r, x, k, h, n): setof[nat] =
  { p | a(x, k, h, n) <= p AND p < r AND f(x, p, h) = f(x, a(x, k, h, n), h) }

% Proving the definitions of N and M are equal
N_eq_M: LEMMA
  x < n AND k <= n IMPLIES N(a(x + 1, k, h, n) + 1, x, k, h, n) = M(x, k, h)

% Inductive definition of N
N_inductive: LEMMA
  x < n AND k <= n IMPLIES
    N(r + 1, x, k, h, n) = IF f(x, r, h) = f(x, a(x, k, h, n), h)
                            THEN add(r, N(r, x, k, h, n))
                            ELSE N(r, x, k, h, n)
                          ENDIF


% Start of final "segment"
u(k, h, n): nat =
  min({ x | a(x, k + 1, h, n) = k })

u_bounded: LEMMA
  u(k, h, n) <= n

% Auxiliary lemma for u_iff_f
u_iff_a: LEMMA
  x < n IMPLIES
    (x < u(k, h, n) IFF a(x, k + 1, h, n) < k)

% Auxiliary lemma for u_iff_f
a_iff_f: LEMMA
  x < n IMPLIES
    (a(x, k + 1, h, n) < k IFF f(x, a(x, k, h, n), h) <= f(x, k, h))

% The righthand-side of this lemma is the test from LinearSearch, while the
% lefthand-side is the conclusion to be drawn from LinearSearch.
% whh426 (5)
u_iff_f: LEMMA
  x < n IMPLIES
    (x < u(k, h, n) IFF f(x, a(x, k, h, n), h) <= f(x, k, h))


% Auxiliary lemma for a_inductive
a_inductive_if_f: LEMMA
  x < n IMPLIES
    f(x, a(x, k, h, n), h) <= f(x, k, h) IMPLIES
      a(x, k + 1, h, n) = a(x, k, h, n)

% Inductive definition of a
% whh426 (6)
a_inductive: LEMMA
  x < n IMPLIES
    a(x, k + 1, h, n) = IF x < u(k, h, n) THEN a(x, k, h, n) ELSE k ENDIF


% Solution for x: f(x, p, h) <= f(x, q, h)
% The context p < q is added to the definition, but should always be true
g(p, q, h): int =
  IF p < q
    THEN floor( (q * q - p * p + h(q) - h(p)) / (2 * (q - p)) )
    ELSE 0
  ENDIF

% Auxiliary lemma for u_eq_g
% The required context for using g is added by "p < q IMPLIES"
% whh426 (7)
f_iff_g: LEMMA
  p < q IMPLIES
    (f(x, p, h) <= f(x, q, h) IFF x <= g(p, q, h))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Properties of datastructure %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Select only the minimal amount of points of a, so that it is represented
% by a (strictly) increasing set. The enumeration of this set is stored as a
% stack in the algorithm, so the elements of Q are referred to as stack values.
% Slight deviation of whh426: Q(k, h, n)(0), simplifying the definition of t
Q(k, h, n): setof[nat] =
  { x | x = 0 OR 1 <= x AND x < n AND a(x - 1, k, h, n) < a(x, k, h, n) }

% A is strictly increasing in its first argument, IF y in Q
```

```
    Q_monotonic_general: LEMMA
      x < y AND Q(k, h, n)(y) IMPLIES a(x, k, h, n) < a(y, k, h, n)

    % Inductive definition of Q
225 Q_inductive: LEMMA
      Q(k + 1, h, n) = { x | x < u(k, h, n) AND Q(k, h, n)(x) OR
                             x < n AND x = u(k, h, n) }

    % Counting the elements in Q
230 q(k, h, n): nat =
      card(Q(k, h, n))

    q_positive: LEMMA
      q(k, h, n) > 0
235
    q_one: LEMMA
      q(1, h, n) = 1

    u_zero_q_one: LEMMA
240   u(k, h, n) = 0 IMPLIES q(k + 1, h, n) = 1

    q_bounded: LEMMA
      q(k, h, n) <= n

245
    % Enumerating the elements of Q in increasing order, with the ith_element
    % theory. By doing so, the function t corresponds with the stack t in the
    % algorithm.
    t(i, k, h, n): nat =
250   ith_element(i, Q(k, h, n), n)

    % The ith_element theory makes proofs with t quite laborious, therefore I added
    % some required basic properties to the theory

255 t_in_Q: LEMMA
      i < q(k, h, n) IMPLIES
        Q(k, h, n)(t(i, k, h, n))

    t_bounded: LEMMA
260   t(i, k, h, n) <= n

    t_bounded_strong: LEMMA
      i < q(k, h, n) IMPLIES
        t(i, k, h, n) < n
265
    t_overflow: LEMMA
      i >= q(k, h, n) IMPLIES
        t(i, k, h, n) = n

270 % whh426 (8)
    t_increasing: LEMMA
      i < q(k, h, n) IMPLIES
        t(i, k, h, n) < t(i + 1, k, h, n)

275 t_increasing_general: LEMMA
      i < j AND j < q(k, h, n) IMPLIES
        t(i, k, h, n) < t(j, k, h, n)

    t_exists: LEMMA
280   Q(k, h, n)(x) IMPLIES
        EXISTS (i): i < q(k, h, n) AND x = t(i, k, h, n)

    t_zero: LEMMA
      t(0, k, h, n) = 0
285
    t_positive: LEMMA
      i > 0 IMPLIES
        t(i, k, h, n) > 0

290
    % Value of a, at points indexed by t(i). The function at corresponds with the
    % stack at in the algorithm.
    at(i, k, h, n): nat =
      a(t(i, k, h, n), k, h, n)
295
    % at is monotonically increasing in the first argument
    % whh426 (8)
    at_increasing: LEMMA
      i + 1 < q(k, h, n) IMPLIES
300     at(i, k, h, n) < at(i + 1, k, h, n)

    % The function value of a for x between t(i) and t(i + 1) is equal to at(i)
    % whh426 (8)
```

```
    a_eq_at: LEMMA
305   i < q(k, h, n) AND t(i, k, h, n) <= x AND x < t(i + 1, k, h, n) IMPLIES
        a(x, k, h, n) = at(i, k, h, n)


    % Application of lemma u_iff_f to stack values
310 % whh426 (9)
    tu_iff_f: LEMMA
      i < q(k, h, n) IMPLIES
        ( t(i, k, h, n) < u(k, h, n) IFF
          f(t(i, k, h, n), at(i, k, h, n), h) <= f(t(i, k, h, n), k, h) )
315
    % maxI is the biggest index where t(i) < u(k), or if no such index exists: 0.
    % Note that the condition "0 < u(k, h, n)" is part of the definition and not
    % added context. Therefore, maxI can be used regardless of this condition.
320 % whh426 mentions a variable j with the same properties, but provides no exact
    % definition
    maxI(k, h, n): nat =
      IF 0 < u(k, h, n)
        THEN max({ i | i < q(k, h, n) AND t(i, k, h, n) < u(k, h, n) })
325     ELSE 0
      ENDIF

    maxI_bounded: LEMMA
      maxI(k, h, n) < q(k, h, n)
330
    % The condition 0 < u(k) is part of the update pre-condition. The remainder of
    % the theory is about finding the correct values to update with in that part
    % of the algorithm.

335 % The range of u(k), useful for applying a_eq_at to u(k)
    u_range: LEMMA
      0 < u(k, h, n) IMPLIES
        t(maxI(k, h, n), k, h, n) < u(k, h, n) AND
        u(k, h, n) <= t(maxI(k, h, n) + 1, k, h, n)
340
    % Auxiliary lemma for u_eq_g
    u_leq_g: LEMMA
      0 < u(k, h, n) AND t(maxI(k, h, n), k, h, n) <= x AND
      x < t(maxI(k, h, n) + 1, k, h, n) IMPLIES
345     x < u(k, h, n) IMPLIES x <= g(at(maxI(k, h, n), k, h, n), k, h)

    % Auxiliary lemma for u_eq_g
    u_leq_g1: LEMMA
      0 < u(k, h, n) IMPLIES
350     u(k, h, n) <= 1 + g(at(maxI(k, h, n), k, h, n), k, h)

    % Auxiliary lemma for u_eq_g
    u_geq_g: LEMMA
      u(k, h, n) < n IMPLIES 1 + g(at(maxI(k, h, n), k, h, n), k, h) <= u(k, h, n)
355
    % The exact value of u(k), defined by the (directly computable) g
    % whh426 provides this equation, only not enumerated
    u_eq_g: LEMMA
      0 < u(k, h, n) IMPLIES
360     u(k, h, n) = min(n, 1 + g(at(maxI(k, h, n), k, h, n), k, h))

    % maxI is an application of the more general absMaxI, defined in the
    % ith_element_theory.
    maxI_abstracted: LEMMA
365   0 < u(k, h, n) IMPLIES
        maxI(k, h, n) = absMaxI(Q(k, h, n), n, u(k, h, n))

    % Auxiliary lemma for q_inductive, the set defined here is a subset of the one
    % defined in Q_inductive
370 maxI_card: LEMMA
      0 < u(k, h, n) IMPLIES
        maxI(k, h, n) = card({ x | x < u(k, h, n) AND Q(k, h, n)(x)}) - 1

    % The inductive definition for q
375 q_inductive: LEMMA
      0 < u(k, h, n) IMPLIES
        q(k + 1, h, n) - 1 = maxI(k, h, n) + IF u(k, h, n) < n THEN 1 ELSE 0 ENDIF

    % Using the property "i <= maxI IMPLIES t(i) < u(k)" and a_inductive, it can
380 % be concluded that the data before t(maxI) is unchanged. Therefore, the stacks
    % are the same as well.
    t_inductive_bounded: LEMMA
      0 < u(k, h, n) AND i <= maxI(k, h, n) IMPLIES
        t(i, k + 1, h, n) = t(i, k, h, n)
385
    % Getting the added element of
```

```
t_inductive_newElement: LEMMA
  0 < u(k, h, n) AND u(k, h, n) < n IMPLIES
    t(maxI(k, h, n) + 1, k + 1, h, n) = u(k, h, n)

% Analogue to the reasoning of t_inductive_bounded, at remains unchanged before
% maxI
at_inductive_bounded: LEMMA
  0 < u(k, h, n) AND i <= maxI(k, h, n) IMPLIES
    at(i, k + 1, h, n) = at(i, k, h, n)

END EFT
```

A.4 programs.pvs

The file programs.pvs contains the implementation of the Program Correctness theory: the Hoare triples.

programs.pvs

```
% W.H. Hesselink
% First version 6th August 2003. Last modification 11 October 2006

% Last modification (by Sebastian Verschoor) 25 July 2012:
%   Added composition_associative, tcHoareIfThenElse, ccHoareIfThenElse

programs [state: TYPE]: THEORY
BEGIN

   program: TYPE = pred[[state, lift[state]]]
   prog, body, progA, progB, progC: VAR program
   x, y, z: VAR state
   yy, zz: VAR lift[state]

% prog(x, yy) means: if prog starts in x it may result in yy,
% where yy = bottom means nontermination

   guard, inv, p0, p1, pre, post: VAR pred[state]
   n, i: VAR nat
   t: VAR int
   vf: VAR [state -> int]

   AND (p0, p1): pred[state] = {x | p0 (x) AND p1(x)}
   NOT (p0): pred[state] = {x | NOT p0 (x)}

% NOTE that these functions AND and NOT are lifted versions of the boolean
% operators with the same names. Therefore you may have to expand them at
% unexpected points! Moreover, this AND can serve as an infix operator.

   ifThenElse (guard, progA, progB): program =
      { (x, yy) |
         IF guard (x) THEN progA (x,yy) ELSE progB (x, yy) ENDIF } ;

   ^ (progA, progB): program =   % sequential composition (infix)
      { (x, yy) |
         yy = bottom and progA (x, yy)
         OR (EXISTS (z): progA (x, up(z)) and progB (z, yy)) }

   composition_associative: LEMMA
     progA ^ (progB ^ progC) = (progA ^ progB) ^ progC

   while (guard, body): program =
      { (x, yy) | EXISTS (ss: sequence[state]):
         ss(0) = x and
         ((EXISTS (n):
             (FORALL (i): i< n implies
                guard (ss (i)) and body (ss(i), up(ss(i+1))) )
             and ((not guard (ss(n)) and yy = up (ss (n)))
                   or (guard (ss(n)) and body (ss (n), bottom)
                       and yy = bottom ) ) )
          or (yy = bottom and
              FORALL (i): guard (ss (i)) and
                          body (ss(i), up(ss(i+1))) ) )  }

   tcHoare (pre, prog, post): bool = % total correctness Hoare triple
      (FORALL (x): pre (x) IMPLIES
         NOT prog (x, bottom) AND
         FORALL (y): prog (x, up (y)) IMPLIES post (y) )

   up_injective: LEMMA (up (x) = up (y) IMPLIES x = y)
```

```
    tcHoareWeakening: LEMMA
       tcHoare (p0, prog, p1) AND subset?(pre, p0) AND subset?(p1, post)
       IMPLIES tcHoare (pre, prog, post)
65
    tcHoareIfThenElse: LEMMA
       tcHoare (pre AND guard, progA, post) AND
       tcHoare (pre AND NOT guard, progB, post)
       IMPLIES tcHoare (pre, ifThenElse(guard, progA, progB), post)
70
    tcHoareComposition: LEMMA
       tcHoare (pre, progA, p0) AND tcHoare (p0, progB, post)
       IMPLIES tcHoare (pre, progA ^ progB, post)

75  isVariant (vf, inv, guard, body): bool =
       (FORALL (x): inv (x) AND guard (x) IMPLIES
          vf (x) >= 0 AND
          FORALL (y): body (x, up (y)) IMPLIES vf (y) < vf (x) )

80  whileTheorem: THEOREM
       tcHoare (inv and guard, body, inv)
       AND isVariant (vf, inv, guard, body)
       IMPLIES tcHoare (inv, while (guard, body), inv and not (guard))

85 % A special theory for deterministic, terminating programs: "commands"

    command: TYPE = [state -> state]
    com, comA, comB: VAR command

90  lift (com): program = { (x, yy) | yy = up (com (x)) }

    tcHoare (pre, com, post): bool =
       (FORALL (x): pre (x) IMPLIES post (com (x)))

95  com_prog_tcHoare: LEMMA
       tcHoare (pre, com, post) IMPLIES tcHoare (pre, lift (com), post)

    isVariant (vf, inv, guard, com): bool =
       (FORALL (x): inv (x) AND guard (x) IMPLIES
100       vf (x) >= 0 AND vf (com (x)) < vf (x) )

    com_prog_isVariant: LEMMA
       isVariant (vf, inv, guard, com) IMPLIES
       isVariant (vf, inv, guard, lift (com))
105
    ifThenElse (guard, comA, comB): command =
       {x | IF guard (x) THEN comA (x) ELSE comB (x) ENDIF } ;

    ^ (comA, comB): command = comB o comA
110 % Sequential composition, see prelude. Note the reversal.

    com_prog_ifThenElse: LEMMA
       lift (ifThenElse (guard, comA, comB)) =
          ifThenElse (guard, lift (comA), lift (comB))
115
    com_prog_composition: LEMMA
       lift (comA ^ comB) = lift (comA) ^ lift (comB)

   % Conditional correctness allows nontermination
120
    ccHoare (pre, prog, post): boolean =
       (FORALL (x, y): pre (x) AND prog (x, up (y)) IMPLIES post (y))

    ccHoare_implied: LEMMA
125    tcHoare (pre, prog, post) IMPLIES ccHoare (pre, prog, post)

    com_prog_ccHoare: LEMMA
       tcHoare (pre, com, post) IMPLIES ccHoare (pre, lift (com), post)

130 ccHoareWeakening: LEMMA
       ccHoare (p0, prog, p1) AND subset?(pre, p0) AND subset?(p1, post)
       IMPLIES ccHoare (pre, prog, post)

    mixedHoareConjunction: LEMMA
135    tcHoare (pre, prog, post) AND ccHoare (p0, prog, p1)
       IMPLIES tcHoare (pre AND p0, prog, post AND p1)

    ccHoareIfThenElse: LEMMA
       ccHoare (pre AND guard, progA, post) AND
140    ccHoare (pre AND NOT guard, progB, post)
       IMPLIES ccHoare (pre, ifThenElse(guard, progA, progB), post)

    ccHoareComposition: LEMMA
       ccHoare (pre, progA, p0) AND ccHoare (p0, progB, post)
```

```
145         IMPLIES ccHoare (pre, progA ^ progB, post)

    ccWhileTheorem: THEOREM
        ccHoare (inv and guard, body, inv)
        IMPLIES ccHoare (inv, while (guard, body), inv and not guard)
150
    boundedBy (vf, t): pred[state] = {x | vf (x) <= t}

    isVariant_implied: LEMMA
        (FORALL (x): inv (x) AND guard (x) IMPLIES vf (x) >= 0) AND
155     (FORALL (n): ccHoare (inv AND guard AND boundedBy (vf, n),
                              body, boundedBy (vf, n - 1) ) )
        IMPLIES isVariant (vf, inv, guard, body)
END programs
```

A.5 ith_element_theory.pvs

The file ith_element_theory.pvs contains the theory that allows elements of a set to be enumerated in ascending order.

ith_element_theory.pvs

```
    % Defining the enumeration of elements in a set of natural numbers
    %
    % Author: Sebastian Verschoor
    % Last modified: 26 July 2012
5
    ith_element_theory: THEORY

    BEGIN

10  IMPORTING max_nat, auxilCard

    % Index values
    i, j: VAR nat

15  % Sets of natural numbers
    U, V: VAR finite_set[nat]

    % Upper bound of the set
    upb: VAR nat
20
    % Element to be added to te set
    newElement: VAR nat

    % Set elements
25  x, y: VAR nat


    % Accessing the elements of set U with recursion. A low index represents a small
    % element in U. If initially i >= card(U), the upperbound (upb) is returned. For
30  % the definition to work, the chosen upb must be bigger than all elements in U.
    ith_element(i, U, upb): RECURSIVE nat =
      IF i = 0
        THEN min(add(upb, U))
        ELSE ith_element(i-1, remove(min(add(upb, U)), U), upb)
35    ENDIF
    MEASURE i

    % Basic properties of ith_element

40  ith_element_in_U: LEMMA
      (FORALL (x): U(x) IMPLIES x < upb) AND i < card(U) IMPLIES
        U(ith_element(i, U, upb))

    ith_element_bounded: LEMMA
45    (FORALL (x): U(x) IMPLIES x < upb) IMPLIES
        ith_element(i, U, upb) <= upb

    ith_element_bounded_strong: LEMMA
      (FORALL (x): U(x) IMPLIES x < upb) AND i < card(U) IMPLIES
50      ith_element(i, U, upb) < upb

    ith_element_overflow: LEMMA
      (FORALL (x): U(x) IMPLIES x < upb) AND i >= card(U) IMPLIES
        ith_element(i, U, upb) = upb
55
    ith_element_increasing: LEMMA
      (FORALL (x): U(x) IMPLIES x < upb) AND i < card(U) IMPLIES
```

```
    ith_element(i, U, upb) < ith_element(i+1, U, upb)

ith_element_increasing_general: LEMMA
  (FORALL (x): U(x) IMPLIES x < upb) AND i < j AND j < card(U) IMPLIES
    ith_element(i, U, upb) < ith_element(j, U, upb)

ith_element_exists: LEMMA
  (FORALL (x): U(x) IMPLIES x < upb) AND U(y) IMPLIES
    EXISTS (i): i < card(U) AND y = ith_element(i, U, upb)


% A new set (V) is constructed from U by introducing newElement. All x in U
% wherefor x < newElement remain in V, while newElement is added to V.

% All elements x < newElement remain the same, thus so does ith_element
new_ith_element_bounded: LEMMA
  (FORALL (x): U(x) IMPLIES x < upb) AND newElement <= upb IMPLIES
    LET V = { x | x < newElement AND U(x) } IN
      i < card(V) IMPLIES ith_element(i, V, upb) = ith_element(i, U, upb)

% absMaxI is the last index where x < newElement
absMaxI(U, upb, newElement): nat =
  max({ i | i < card(U) AND ith_element(i, U, upb) < newElement })

% Two definitions for the new set, using newElement and using absMaxI
equalsets: LEMMA
  (FORALL (x): U(x) IMPLIES x < upb) AND
    0 < newElement AND newElement <= upb AND U(0) IMPLIES
      { x | U(x) AND x < newElement } =
      { x | EXISTS (i): ith_element(i, U, upb) = x AND
            i <= absMaxI(U, upb, newElement) }

% The relationship between the number of elements in the new set and absMaxI
equalsets_card: LEMMA
  (FORALL (x): U(x) IMPLIES x < upb) AND
    0 < newElement AND newElement <= upb AND U(0) IMPLIES
      card({ x | U(x) AND x < newElement }) = absMaxI(U, upb, newElement) + 1

% Adding newElement, bigger than all elements in U, is indexed by card(U)
ith_element_add_newElement: LEMMA
  (FORALL (x): U(x) IMPLIES x < newElement) AND newElement < upb IMPLIES
    ith_element(card(U), add(newElement, U), upb) = newElement

% Applying the above lemma to
new_ith_element_add_newElement: LEMMA
  (FORALL (x): U(x) IMPLIES x < upb) AND
    0 < newElement AND newElement < upb AND U(0) IMPLIES
      LET V = add(newElement, { x | x < newElement AND U(x) }) IN
        ith_element(absMaxI(V, upb, newElement) + 1, V, upb) = newElement

END ith_element_theory
```

A.6 more_floor.pvs

The file more_floor.pvs contains the theory for proving with the floor- or div-function.

more_floor.pvs

```
% Wim H. Hesselink, September 2005
% A slight extension of the library prelude.pvs

more_floor: THEORY

BEGIN
  n, m: VAR int
  x, y: VAR real
  b: VAR posnat

  floor_Galois: THEOREM
    (n <= floor(x)) = n <= x

  floor_monotonic: LEMMA
    x <= y IMPLIES floor(x) <= floor(y)

  ndiv_floor: LEMMA
    ndiv(n, b) = floor(n/b)

  ndiv_Galois: LEMMA
    (m <= ndiv(n, b)) = (m * b <= n)
```

```
  ndiv_monotonic: LEMMA
    m <= n IMPLIES ndiv(m, b) <= ndiv(n, b)

  ndiv_mult: LEMMA
    ndiv(b * n, b) = n

END more_floor
```

A.7 max_nat.pvs

The file max_nat.pvs makes it possible to take the maximum element of a finite set of natural numbers.

max_nat.pvs

```
% Getting the maximum value of a set of natural numbers
%
% Author: Sebastian Verschoor
% Last modified: 26 July 2012

max_nat: THEORY

BEGIN

% PVS declarations
S: VAR finite_set[nat]
x: VAR nat

max(S): RECURSIVE nat =
  IF empty?(S) THEN 0 ELSE
    LET ms = min(S),
        ss = remove(ms, S) IN
      IF empty?(ss) THEN ms ELSE max(ss) ENDIF
  ENDIF
MEASURE card(S)

% While the definition of max contains a case for getting the maximum natural
% number of an empty set, this actually makes no sense. Therefore, the max_lemma
% adds the requirement that max is only applied to nonempty sets.
max_lemma: LEMMA
  nonempty?(S) IMPLIES S(max(S)) AND (FORALL x: S(x) IMPLIES x <= max(S))

END max_nat
```

A.8 auxilCard.pvs

The file auxilCard.pvs makes it possible to count the elements in the image of an injective function.

auxilCard.pvs

```
% Extension of the prelude file:
%   counting elements in the image of an injective function
%
% Author: Wim Hesselink
% Last modified: 24 July 2012

auxilCard[T: TYPE]: THEORY

BEGIN
  f: VAR [nat -> T]
  i, n: VAR nat
  t: VAR T

  image(f, n)(t): bool =
    EXISTS i: i < n AND f(i) = t

  inverse(f, n)(t: (image(f, n))): below(n) =
    epsilon({i | i < n AND f(i) = t})

  inverse_injectiveW: LEMMA
    injective?(inverse(f, n))

  is_finite_image: LEMMA
    is_finite(image(f, n))

```

```
  inverse_bijectiveW: LEMMA
    injective?(restrict[nat, below(n), T](f))
    IMPLIES bijective?(inverse(f, n))

30 card_image: LEMMA
    injective?(restrict[nat, below(n), T](f))
    IMPLIES is_finite(image(f, n)) AND card(image(f, n)) = n

END auxilCard
```