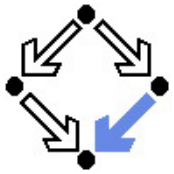

JOHANNES KEPLER UNIVERSITY LINZ
RESEARCH INSTITUTE FOR SYMBOLIC COMPUTATION

Technical Report

Proof Based Synthesis of Sorting Algorithms

Authors:
Isabela Drămnesc and Tudor Jebelean

July, 2010

We present some case studies in constructive synthesis of sorting algorithms. In order to synthesize some algorithms on tuples (like e. g. insertion-sort, merge-sort) we use an approach based on proving. Namely, we start from the specification of the problem (input and output condition) and we construct an inductive proof of the fact that for each input there exists a solution which satisfies the output condition. The problem will be reduced into smaller and smaller problems, the method will be applied like in a "cascade" and finally the problem is so simple that the corresponding algorithm (function) already exists in the knowledge. The algorithm can be then extracted immediately from the proof.

These experiments are paralleled with the exploration of the appropriate theory of tuples. The purpose of these experiments is multi-fold: to construct the appropriate knowledge base necessary for this type of proofs, to find the natural deduction inference rules and the necessary strategies for their application, and finally to implement the corresponding provers in the frame of the *Theorema* system.

The novel specific feature of our approach is applying this method like in a "cascade" and the use (as much as possible) of first order predicate logic because this increases the feasibility of proving.

Contents

Chapter 1

Introduction

Mathematics is a technique of solving problems and the essence of mathematics is proving. By reasoning about the problems we obtain knowledge. This knowledge can be managed for theory exploration and reasoning about the knowledge can be implemented on computers.

In this paper we present some case studies on synthesis of sorting algorithms (like e.g. insertion-sort, merge-sort). We use the constructive synthesis, but the method for proving that we apply like in a "cascade" is new. In chapter 4 we describe in general the method and we apply this method like in a "cascade" on the sorting algorithms, in particular on the Insertion-Sort and on the Merge-Sort algorithms for tuples.

1.1 Theory Exploration

Mathematical theories (on computer or not) are built incrementally starting from a set of axioms, definitions and by adding propositions which we check and prove. A mathematical theory is expressed by the language, the knowledge base and the inference rules. The *language "$\mathcal{L}$"* contains a set of *predicate, function, and constant* symbols. The *knowledge base "$\mathcal{KB}$"* contains a set of predicate logic formulae over the language: *axioms, theorems, properties*. The *inference rules "$\mathcal{IR}$"* are rules that describe the reasoning system for the theory. The rules describe how to transform the current proof situation into a new one.

$$Theory = \langle \mathcal{L}, \mathcal{KB}, \mathcal{IR} \rangle$$

In [3], the author talks about automated theorem proving versus mathematical theory exploration using computers and explains that automated theorem proving does not mean only to prove a single theorem, it means that by making the proof we are doing the process of exploring theory using computers. And he explains all the steps that one should follow in the process of exploration using the *Theorema* system.

1.2 Related work

Bruno Buchberger introduces some strategies for systematic theory exploration in a bottom-up and in a top-down way, see [3]. In fact many of the definitions and assumptions from tuple theory which we use in this paper are developed in that research work. Our previous work on bottom-up exploration of tuple theory is presented in [13].

Program synthesis in computational logic is a well studied field. For an overview of several approaches that have been investigated see [2]. A tutorial on synthesis of programs one can find in [19].

Deductive techniques for program synthesis are presented in [20] and techniques for constructing induction rules for deductive synthesis proofs are presented in [8].

Bruno Buchberger introduces a method for algorithm synthesis, called "lazy thinking", see [4] and also a model for theory exploration using schemes, see [5]. The "lazy thinking" method is implemented in the TH∃OREM∀ system [7], see [6]. In this approach, a "program schemata" (the structure of the solving term) is given to the synthesis procedure. In contrast, in our approach this schemata is also discovered by the procedure, only the inductive principle of the domain is given. Another approach for automatize the synthesis of logic programs was introduced in [17].

A case study on exploration of natural numbers based on schemes (a bottom-up exploration) is done in [14] and also the decomposition of natural numbers (top-down exploration) in [11].

In [12] the author showed how to use program transformation techniques to develop versions of six well-known sorting algorithms. For a classification of the sorting algorithms see [18]. An alternative taxonomy to the one from [16] of the sorting algorithms was introduced in [21].

The synthesis of sorting algorithms on tuples in higher order predicate logic, using the "lazy thinking" method you can find in [10]. A case study on synthesis of the Merge-Sort algorithm one can find in [22], [23].

In this paper we explore the Tuple Theory in a top-down way using a method for proving in the context of constructive synthesis that is applied like in a "cascade" to all the reduced problems.

In contrast with the other existent case studies on sorting algorithms from literature, in particular sorting of tuples, in our approach we use a different method for synthesize the algorithms and the use as much as possible of first order logic. This method was first introduced in 2008, for details see [9]. A case study of transforming recursive procedures into tail recursive using this method of synthesis one can find in [1].

The process of constructing mathematical model, designing an algorithm and the elaboration of a few versions using the Lisp program for the reverse of a list is done in [15]. It starts from a problem that for all lists L there exists a list L' such that L and L' are in the relation $\mathbf{R}$. And all that we know is that there exists a function $R[L]$ with the property that L and $R[L]$ are in the relation $\mathbf{R}$. Then, he adds some properties for the function R and he illustrates the characteristics of the program styles: declarative, functional, tail recursive and imperative. So, when we write a program we need to construct a mathematical model and to design an algorithm that is correct and

efficient.

1.3 Structure of the paper

In chapter 2 we give a short introduction on exploration of Tuple Theory and on algorithm synthesis.

In chapter 3 we present some case studies on synthesis of the insertion-sort and the merge-sort algorithms in the case when we guess the witness during the proof. We start from the specification of the problem (input and output conditions), generate a conjecture, construct an inductive proof, guess the witness and then check the witness in the goals. During the proof we add propositions into the knowledge base, prove them and then use them in the proof. In this way we explore the appropriate theory of tuples. This way of guessing the witness is helpful for the second case study on synthesis of sorting algorithms on tuples that we present in chapter 4.

In chapter 4 we present in general the method that we apply for synthesize some algorithms on tuples, in particular the insertion-sort and the merge-sort algorithms and some experiments on synthesis of the insertion-sort and the merge-sort algorithms in the case when we try to find the witness during the proof. We start from the specification of the problem (input and output conditions), generate a conjecture, construct an inductive proof and try to find the witness that satisfies the output conditions. During the proof we add propositions into the knowledge base, prove them and then use them in the proof. In this way we explore the appropriate theory of tuples.

In chapter 5 we present conclusions and future plans for research.

Chapter 2

Exploration and Synthesis

2.1 Exploration of Tuple Theory

The Tuple Theory contains the language, the knowledge base and the inference rules:

The Language $\mathcal{L}$ contains a set of predicates, functions, constant symbols:

$$\mathcal{L}_{\mathcal{T}} = \left\langle \langle IT, = \rangle, \langle \smile, Id \rangle, \langle \langle \rangle \rangle \right\rangle,$$

where the predicates are: the unary predicate "IT" that describes the type of the objects from the theory ("IT" stands for "Is Tuple") and the binary equality "=" predicate; the functions are: the binary function "$\smile$" that adds an element at the beginning of a tuple and the unary identity "Id" function and as a constant we consider the empty tuple represented by $\langle\rangle$.

The Knowledge Base that contains a collection of known concepts, a collection of properties of the known concept and formulae that describe the known notions and the relation between known concepts and new concepts. The new notions can be: new concepts, new notions introduced by definitions, a collection of goals.

The inference rules are rules from predicate logic, rewriting rules and rules from the dependent domain.

In our notation, $\langle\rangle$ represents the empty tuple, $a \smile \langle\rangle$ is a tuple having a single element a, and $a \smile X$ is the tuple having head a and tail X.

2.1.1 Bottom-up Exploration

In the bottom-up exploration we start from a set of axioms and at each step we introduce new notions (definitions, propositions). This process is not very useful because at each step we have to make a complete exploration, so our knowledge base is consistent, but when we want to prove something we don't use all the knowledge that we have, so we can waste time.

In more detail you can see the bottom-up exploration of Tuple Theory in the *Theorema* system, see [7], in [13]. Starting from two axioms of "generation" and "unicity" add propositions we built the entire Tuple Theory. All the new notions introduced are checked with the "Compute" command from the *Theorema* system and for proving propositions there are used some of the existent provers from the system and the prover created for the Tuple Theory, "TuplesProverTM".

2.1.2 Top-down Exploration

In the top-down exploration we start from a problem and we try to prove the solution. Usually, this proof will fail because we do not have sufficient knowledge base and from that, at each failing we introduce new notions (propositions) which we use in the next proofs. After each failing proof we obtain a proposition that we consider true, we prove it and we use it in the proof.

We repeat this process until we prove the goal. But there are cases when this process does not finish.

2.2 Algorithm Synthesis

In program synthesis we deal with the problem of finding an algorithm starting from a specification. The concern of program synthesis is to develop methods and tools for mechanize or automate (parts of) the process of finding an algorithm that satisfies the specification. Although constructive logic already gives comprehensive methods for extracting algorithms from proofs, it is still a challenge to actually find such proofs for concrete problems.

In program synthesis in computational logic we deal with the question "Given a specification how can we find an executable program that satisfies the specification?". The specification (input and output conditions) is an expression from a certain language and it has to describe what the program should do. The specification can be complete or incomplete. If we have an incomplete specification then the automatization of the synthesis process cannot be completely done. The specification, the developed program and the relation(s) between them can all be expressed in the same logic.

2.2.1 Methods

There exist many different methods for synthesize programs like constructive synthesis, deductive synthesis, schema-based synthesis, inductive synthesis. In constructive synthesis we generate a conjecture from the specification, prove this conjecture and from the proof extract the algorithm. In deductive synthesis we directly obtain the program by a suitable transformation of the specification. In schema-based synthesis one use program schemas that guide the entire process of synthesis. In inductive synthesis we induce the program from an incomplete specification. For an overview of these several methods for synthesize recursive programs in computational logic see [2].

In this paper we use the constructive synthesis, but the method for proving that we apply like in a "cascade" is new. In chapter 3 we describe in general the method and the way we apply this method like in a "cascade" on the sorting algorithms, in particular on the Insertion-Sort and on the Merge-Sort algorithm for tuples.

These experiments are paralleled with the exploration of the appropriate theory of tuples. The purpose of these experiments is multi-fold: to construct the appropriate knowledge base necessary for this type of proofs, to find the natural deduction inference rules and the necessary strategies for their application, and finally to implement the corresponding provers in the frame of the *Theorema* system.

Chapter 3

Synthesis of Sorting Algorithms on Tuples-the case when we guess the witness

3.1 Description of the method

We start from the specification of the problem (input and output condition). Given the *Problem Specification:*

Input: $I_F[X]$

Output: $O_F[X, Y]$

guess the definition of F such that $\underset{X:I_F}{\forall} O_F[X, F[X]]$. We use square brackets as in $f[x]$ for function application and for predicate application, instead of the usual round paran-theses as in $f(x)$. Also, the quantified formula above is a notation for: $(\forall X)(I_F[X] \Longrightarrow O_F[X, F[X]])$.

We prove $\underset{X:I_F}{\forall} \underset{Y}{\exists} O_F[X, Y]$, either directly or using an induction principle (the choice of the proof style and of the induction principle is not automatic).

Direct proof. We take X_0 arbitrary but fixed (a new constant), we assume $I_F[X_0]$ and by proof we guess a witness $T[X_0]$ (a term depending on X_0) such that $O_F[X_0, T[X_0]]$. In this case, the algorithm is $F[X] = T[X]$.

Inductive proof. Let us consider for illustration the head-tail inductive definition of tuples and that the functions Head and Tail are in the knowledge base. Moreover, in order for this induction scheme to be appropriate for our algorithm, one must have the properties: $I_F[\langle\rangle]$, and $\underset{a}{\forall}\underset{X}{\forall} I_F[a \frown X] \Rightarrow I_F[X]$. (That is: the input condition must hold for the base case and for the inductive decomposition of the argument.)

Base case: We prove $\underset{Y}{\exists} O_F[\langle\rangle, Y]$. The proof succeeds if we guess a ground term C such that $O_F[\langle\rangle, C]$, and then we know that $F[\langle\rangle] = C$.

Induction step: For arbitrary but fixed a and X_0 (satisfying I_F), we assume $\underset{Y}{\exists} O_F[X_0, Y]$

and we prove $\underset{Y}{\exists} O_F[a \smile X_0, Y]$. We Skolemize the assumption by introducing a new constant Y_0 for the existential Y. The proof succeeds if we guess a witness $T[a, X_0, Y_0]$ (term depending on a, X_0, and Y_0) such that $O_F[a \smile X_0, T[a, X_0, Y_0]]$, and then we know that $F[a \smile X] = T[a, X, F[X]$.

Algorithm extraction: Finally the algorithm is expressed as:

$$F[X] = \begin{cases} C, & \text{if} X = \langle\rangle \\ T[Head[X], Tail[X], F[Tail[X]], & \text{if} X \neq \langle\rangle \end{cases}$$

Cascading. Assume that in the induction step above we cannot guess the correct witness. The we can create the following new problem: "Given a, X, Y such that $I_F[X]$ and $O[X, Y]$, guess a Z such that $O_F[a \smile X, Z]$." That is, if we cannot guess an appropriate term, we try to synthesize (by a possibly different induction principle) a new function $G[a, X, Y]$ which does the job. This new problem looks more complicated, but it is logically simpler, because the arguments fulfill more conditions, while the output requirement is the same as before. This cascading principle can be repeated, and the problem will be reduced into smaller and smaller problems until the necessary functions are already in the knowledge base.

During the proof we discover propositions that we need for the proof in order to succeed. We prove these propositions and after that introduce them into the knowledge and use them in the proof. We will not go into details in this paper for the process of inventing propositions. We just remark that by adding propositions to the knowledge base we explore the theory. Thus the process of proving is paralleled to the process of exploring (building) the theory, and the case study presented here increases our experience in developing a reasonable theory of tuples for further practical applications and it is a introductory step for the method and the case studies from chapter 4.

3.2 Synthesis of INSERTION SORT

3.2.1 CASE I--when we know the function "Insertion"

We know the definitions "≈", "dfo", "◁", "IsSorted", see Appendix.

and also the definition "Insertion":

```
Definition["Insertion", any[a, b, X],
                    Insertion[a, ⟨⟩] = a ⌢ ⟨⟩
      (a ≤ b) ⟹ (Insertion[a, b ⌢ X] = a ⌢ (b ⌢ X))
    (a > b) ⟹ (Insertion[a, b ⌢ X] = b ⌢ Insertion[a, X])]
```

Problem Specification:

$$I_{\text{InsertSort}}[X]: \text{True}$$

$$O_{\text{InsertSort}}[X, Y]: \begin{cases} X \approx Y \\ \text{IsSorted}[Y] \end{cases}$$

We generate the conjecture:

```
Proposition["problem InsertionSort",
  ∀X ∃Y P[X, Y]]
```

where

```
P[X, Y]: { X ≈ Y
         { IsSorted[Y]
```

We prove the proposition "problem InsertionSort" by induction over X:

Base case: X=⟨⟩

$$\text{Prove } \exists_Y P[\langle\rangle, Y]: \exists_Y \begin{cases} \langle\rangle \approx Y \\ \text{IsSorted}[Y] \end{cases}$$

Witness guessed: Y=⟨⟩.

So, we obtain $\mathcal{S}[\langle\rangle]=\langle\rangle$. (*)

$$\text{Prove } \begin{cases} \langle\rangle \approx \langle\rangle \\ \text{IsSorted}[\langle\rangle] \end{cases}$$

This is true by the definition "≈" and by the definition "IsSorted".

Induction Step:

Assume: $\exists_Y P[X_0, Y]: \exists_Y \begin{cases} X_0 \approx Y \\ \text{IsSorted}[Y] \end{cases}$. By Skolemization we know:

$P[X_0, Y_0]$: $\begin{cases} X_0 \approx Y_0 & \text{(H1)} \\ \text{IsSorted}[Y_0] & \text{(H2)} \end{cases}$ and

Prove: $\exists_Y P[a \frown X_0, Y]$: $\exists_Y \begin{cases} a \frown X_0 \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (G)

Witness guessed: Y=Insertion[a, Y_0].

So, we obtain $\mathcal{S}[a \frown X_0]$=Insertion[a, Y_0]. (**)

Prove: P[a$\frown X_0$, Insertion[a, Y_0]]: $\begin{cases} a \frown X_0 \approx \text{Insertion}[a, Y_0] & \text{(G1 .1)} \\ \text{IsSorted}[\text{Insertion}[a, Y_0]] & \text{(G1 .2)} \end{cases}$

Prove (G1.1) $a \frown X_0 \approx \text{Insertion}[a, Y_0] \Longleftrightarrow \begin{cases} a \lhd \text{Insertion}[a, Y_0] & \text{(G1 .11)} \\ X_0 \approx \text{dfo}[a, \text{Insertion}[a, Y_0]] & \text{(G1 .12)} \end{cases}$

For proving (G1.11) and (G1.12) the theory must contain the propositions:

Proposition["IsElem in Insertion", $\forall_{a,X}$ (a $\lhd$ Insertion[a, X])]

Proposition["The same elements in Insertion", $\forall_{a,X,Y}$ ((X $\approx$ Y) $\Longrightarrow$ (X $\approx$ dfo[a, Insertion[a, Y]]))]

By the proposition "IsElem in Insertion" (G1.11) is proved.

By the proposition "The same elements in Insertion" (G1.12) is proved.

We will prove the propositions "IsElem in Insertion" and "The same elements in Insertion" additionally, during the theory exploration.

Prove (G1.2) IsSorted[Insertion[a, Y_0]]

The theory must contain the proposition:

Proposition["Sorting using Insertion", $\forall_{a,X}$ (IsSorted[X] $\Longrightarrow$ IsSorted[Insertion[a, X]])]

By the proposition "Sorting using Insertion" our goal (G1.2) is proved.

We will prove the proposition "Sorting using Insertion" additionally during the theory exploration.

From (*) and (**) and by the transformation rule ($X_0 \longrightarrow X$, $Y_0 \longrightarrow \mathcal{S}$[X]) we obtain the algorithm:

$$\forall_{a,X} \begin{cases} \mathcal{S}[\langle\rangle] = \langle\rangle \\ \mathcal{S}[a \frown X] = \text{Insertion}[a, \mathcal{S}[X]] \end{cases}$$

This is the algorithm corresponding to "Insertion Sort" (InsSort):

$$\underset{a,X}{\forall} \begin{cases} \textbf{InsSort}[\langle\rangle] = \langle\rangle \\ \textbf{InsSort}[a \frown X] = \textbf{Insertion}[a, \textbf{InsSort}[X]] \end{cases}$$

Prove the propositions "IsElem in Insertion", "The same elements in Insertion" and "Sorting using Insertion":

Prove "IsElem in Insertion": $\underset{a,X}{\forall}$ **(a ◅ Insertion[a, X])**

We take a_0 arbitrary, but fixed and we prove by induction over X:

Base case: X=⟨⟩

Prove: $a_0 \triangleleft \text{Insertion}[a_0, \langle\rangle]$. This is true by the definition "Insertion".

Induction Step: X: $b_0 \frown X_0$

We assume: $a_0 \triangleleft \text{Insertion}[a_0, X_0]$ and

prove: $a_0 \triangleleft \text{Insertion}[a_0, b_0 \frown X_0]$ (G2).

We prove this by case distinction using the definition "Insertion":

Case 1: $a_0 \leq b_0$

Prove: $a_0 \triangleleft a_0 \frown (b_0 \frown X_0)$. This is true by the definition "◅".

Case 2: $a_0 > b_0$

Prove: $a_0 \triangleleft b_0 \frown \text{Insertion}[a_0, X_0]$ (G3)

By definition "◅" and by the fact that $a_0 \neq b_0$ we can rewrite (G3)

We must prove: $a_0 \triangleleft \text{Insertion}[a_0, X_0]$ that was our assumption

and we are done.

Prove "The same elements in Insertion":

$\underset{a,X,Y}{\forall}$ **((X ≈ Y) ⟹ (X ≈ dfo[a, Insertion[a, Y]]))**

Take a_0 arbitrary, but fixed and prove by induction over X.

Base case: X=⟨⟩

Prove $\underset{Y}{\forall} ((\langle\rangle \approx Y) \Longrightarrow (\langle\rangle \approx \text{dfo}[a_0, \text{Insertion}[a_0, Y]]))$ (G)

Take Y_0 arbitrary, but fixed and prove:

$(\langle\rangle \approx Y_0) \Longrightarrow (\langle\rangle \approx \text{dfo}[a_0, \text{Insertion}[a_0, Y_0]])$ (G1)

For proving (G1) we:

assume $\langle\rangle \approx Y_0$ (H1) and

prove $\langle\rangle \approx \text{dfo}[a_0, \text{Insertion}[a_0, Y_0]]$ (G2).

By (H1) in (G2) we must prove $\langle\rangle \approx \text{dfo}[a_0, \text{Insertion}[a_0, \langle\rangle]]$ (G3)

By definition "Insertion" (G3) is $\langle\rangle \approx \mathrm{dfo}[a_0,\ a_0 \frown \langle\rangle]$ (G4).

By definition "dfo" this is $\langle\rangle \approx \langle\rangle$ that is true by definition "$\approx$".

Induction Step: X: $b_0 \frown X_0$

Assume:

$\forall_Y ((X_0 \approx Y) \Longrightarrow (X_0 \approx \mathrm{dfo}[a_0, \mathrm{Insertion}[a_0, Y]]))$ (H2).

By Skolemization we know:

$(X_0 \approx Y_0) \Longrightarrow (X_0 \approx \mathrm{dfo}[a_0, \mathrm{Insertion}[a_0, Y_0]])$ (H3)

Prove: $\forall_Y ((b_0 \frown X_0) \approx Y) \Longrightarrow ((b_0 \frown X_0) \approx \mathrm{dfo}[a_0, \mathrm{Insertion}[a_0, Y]])$ (G*)

We prove (G*) by induction over Y:

Base case: $Y = \langle\rangle$

Prove $((b_0 \frown X_0) \approx \langle\rangle) \Longrightarrow ((b_0 \frown X_0) \approx \mathrm{dfo}[a_0, \mathrm{Insertion}[a_0, \langle\rangle]])$ (G*1).

For proving (G*1) we assume: $(b_0 \frown X_0) \approx \langle\rangle$ (H2.1) and

Prove: $(b_0 \frown X_0) \approx \mathrm{dfo}[a_0, \mathrm{Insertion}[a_0, \langle\rangle]]$ (G*2).

By definition "Insertion" (G*2) is $(b_0 \frown X_0) \approx \mathrm{dfo}[a_0,\ a_0 \frown \langle\rangle]$ (G*3).

By definition "dfo" we must prove:

$(b_0 \frown X_0) \approx \langle\rangle$ (G*4) this is true by (H2.1).

Induction Step: Y: $c_0 \frown Y_0$

Assume: $((b_0 \frown X_0) \approx Y_0) \Longrightarrow ((b_0 \frown X_0) \approx \mathrm{dfo}[a_0, \mathrm{Insertion}[a_0, Y_0]])$ (H2.2)

Prove:

$((b_0 \frown X_0) \approx (c_0 \frown Y_0)) \Longrightarrow ((b_0 \frown X_0) \approx \mathrm{dfo}[a_0, \mathrm{Insertion}[a_0, c_0 \frown Y_0]])$ (G*5).

For proving (G*5) we assume: $(b_0 \frown X_0) \approx (c_0 \frown Y_0)$ (H2.3) and

Prove: $(b_0 \frown X_0) \approx \mathrm{dfo}[a_0, \mathrm{Insertion}[a_0, c_0 \frown Y_0]]$ (G*6)

We prove this by case distinction using the definition "Insertion":

Case 1: $a_0 \leq c_0$

Prove: $(b_0 \frown X_0) \approx \mathrm{dfo}[a_0, a_0 \frown (c_0 \frown Y_0)]$ (G*7).

By the definition "dfo" this is: $(b_0 \frown X_0) \approx (c_0 \frown Y_0)$ (G*8) that is true by (H2.3).

Case 2: $a_0 > c_0$

Prove: $(b_0 \frown X_0) \approx \mathrm{dfo}[a_0, c_0 \frown \mathrm{Insertion}[a_0, Y_0]]$ (G*9).

By the def. "dfo" and by the fact that $a_0 \neq c_0$ we have to prove:

$(b_0 \frown X_0) \approx c_0 \frown \mathrm{dfo}[a_0, \mathrm{Insertion}[a_0, Y_0]]$ (G*10).

By the proposition "Delete first occurence in Insertion" we obtain:

$(b_0 \frown X_0) \approx c_0 \frown Y_0$ (G*11) that is true by (H2.3).

So, the proposition "The same elements in Insertion" is proved.

The theory must contain the proposition:

Proposition["Delete first occurence in Insertion", $\forall_{a,X}$ (dfo[a, Insertion[a, X]] = X)]

Prove "Delete first occurence in Insertion":

We take a_0 arbitrary, but fixed and prove by induction over X.

Base case: X=⟨⟩

Prove: dfo[a_0, Insertion[a_0, ⟨⟩]] = ⟨⟩ (G1).

By the definition "Insertion" we obtain: dfo[a_0, a_0⌢⟨⟩]] = ⟨⟩ (G1.1)

that is true by the definition "dfo".

Induction Step: X: b_0 ⌢ X_0

Assume: dfo[a_0, Insertion[a_0, X_0]] = X_0 (H1) and

Prove: dfo[a_0, Insertion[a_0, b_0 ⌢ X_0]] = b_0⌢X_0 (G2).

We prove this by case distrinction using the definition "Insertion":

Case 1: $a_0 \le b_0$

Prove: dfo[a_0, a_0⌢(b_0 ⌢ X_0)] = b_0⌢X_0 (G3).

This is true by the definition "dfo".

Case 2: $a_0 > b_0$

Prove: dfo[a_0, b_0 ⌢Insertion[a_0, X_0]] = b_0⌢X_0 (G4).

By definition "dfo" we obtain: b_0 ⌢dfo[a_0, Insertion[a_0, X_0]] = b_0⌢X_0

that is true by (H1).

So, the proposition "Delete first occurence in Insertion" is proved.

Prove "Sorting using Insertion": $\forall_{a,X}$ (IsSorted[X] ⟹ IsSorted[Insertion[a, X]])

We take a_0 arbitrary, but fixed and prove by induction over X:

Base case: X=⟨⟩

Prove IsSorted[⟨⟩]⟹IsSorted[Insertion[a_0,⟨⟩]] (G1).

For proving (G1) we assume IsSorted[⟨⟩] (H1) and

prove IsSorted[Insertion[a_0,⟨⟩]] (G1.1).

By definition "Insertion" we must prove: IsSorted[a_0⌢⟨⟩] (G1.2).

This is true by the definition "IsSorted".

Induction Step: X: b_0 ⌢ X_0

We assume: IsSorted[X_0]⟹IsSorted[Insertion[a_0, X_0]] (H2) and

Prove: IsSorted[b_0 ⌢ X_0]⟹IsSorted[Insertion[a_0, b_0 ⌢ X_0]] (G2).

For proving (G2) we assume IsSorted[b_0 ⌢ X_0] (H3) and

Prove: IsSorted[Insertion[a_0, $b_0 \frown X_0$]] (G3).

From "property of sort", (H3) by Modus Ponens we know IsSorted[X_0] (H4).

From (H4) and (H2) by Modus Ponens we know IsSorted[Insertion[a_0, X_0]] (H5).

We prove (G3) by case distinction using the definition "Insertion":

Case 1: $a_0 \leq b_0$

Prove: IsSorted[$a_0 \frown (b_0 \frown X_0)$] (G3.1)

By definition "IsSorted" we have to prove IsSorted[$b_0 \frown X_0$] (G3.2)

that was our assumption (H3).

Case 2: $a_0 > b_0$

Prove: IsSorted[$b_0 \frown$ Insertion[a_0, X_0]] (G3.2)

By (H4), (H5) and by the definition "IsSorted" this is true.

So, we proved the proposition "Sorting using Insertion".

The theory must contain the proposition:

$$\text{Proposition}\left["\text{property of sort}", \underset{a,X}{\forall} (\text{IsSorted}[a \frown X] \Longrightarrow \text{IsSorted}[X])\right]$$

Prove the proposition "property of sort":

We take a_0 arbitrary, but fixed and prove by induction over X:

Base case: X=⟨⟩

Prove IsSorted[$a_0 \frown \langle\rangle$]⟹IsSorted[⟨⟩] (G1).

For proving (G1) we assume IsSorted[$a_0 \frown \langle\rangle$] (H1) and prove: IsSorted[⟨⟩] (G1.1). This is true by definition "IsSorted".

Induction step: X: $b_0 \frown X_0$

Assume: IsSorted[$a_0 \frown X_0$]⟹IsSorted[X_0] (H2) and

prove: IsSorted[$a_0 \frown (b_0 \frown X_0)$]⟹IsSorted[$b_0 \frown X_0$] (G2).

For proving (G2) we assume: IsSorted[$a_0 \frown (b_0 \frown X_0)$] (H3) and

prove: IsSorted[$b_0 \frown X_0$] (G3).

By (H3) and the definition "IsSorted" we know:

$a_0 \leq b_0$ (H3.1) ∧ IsSorted[$b_0 \frown X_0$] (H3.2).

So, (G3) is true by (H3.2).

So, we proved the proposition "property of sort".

3.2.2. CASE II--when we do not know the function "Insertion" and we synthetise it

We know the definitions: "$\approx$", "dfo", "$\lhd$", "IsSorted", see Appendix.

Reduced Problem 1:

Specification (X: a,T):

$I_{\text{Ins}}[a, T]$: IsSorted[T]

$O_{\text{Ins}}[a, T, Y] : \begin{cases} a \frown T \approx Y \\ \text{IsSorted}[Y] \end{cases}$

Prove:

$$\underset{T}{\forall} P[T] \iff \underset{T}{\forall}\,\underset{a}{\forall} \left(I_{\text{Ins}}[a, T] \Longrightarrow \underset{Y}{\exists} O_{\text{Ins}}[a, T, Y] \right)$$

For proving we use the Induction Principle Head/Tail (Decomposition)

T: $\begin{cases} \langle\rangle & (P[\langle\rangle]) \\ b \frown T & (P[T] \Longrightarrow P[b \frown T]) \end{cases}$

We take *a* arbitrary, but fixed and we prove:

Base case: T=⟨⟩

Prove P[⟨⟩]: $\left(I_{\text{Ins}}[a, \langle\rangle] \Longrightarrow \underset{Y}{\exists} O_{\text{Ins}}[a, \langle\rangle, Y] \right)$

Assume: $I_{\text{Ins}}[a, \langle\rangle]$: IsSorted[⟨⟩] (H1) and

Prove: $\underset{Y}{\exists} O_{\text{Ins}}[a, \langle\rangle, Y]$: $\underset{Y}{\exists} \begin{cases} a \frown T \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (G1)

Guess witness Y=a⌢⟨⟩.

Thus, we obtain: Insertion[a, ⟨⟩]= a⌢⟨⟩. (▫)

Prove: $O_{\text{Ins}}[a, \langle\rangle, a \frown \langle\rangle]$: $\begin{cases} a \frown \langle\rangle \approx a \frown \langle\rangle & \text{(G1 .1)} \\ \text{IsSorted}[a \frown \langle\rangle] & \text{(G1 .2)} \end{cases}$

(G1.1) is true by the proposition "reflexivity in $\approx$".

(G1.2) is true by the definition "IsSorted".

Induction Step: T:b⌢T

Assume: $I_{\text{Ins}}[a, T] \Longrightarrow \underset{Y}{\exists} O_{\text{Ins}}[a, T, Y]$ that is:

$$\text{IsSorted}[T] \Longrightarrow \underset{Y}{\exists} \begin{cases} a \frown T \approx Y \\ \text{IsSorted}[Y] \end{cases} \quad \text{(H1).}$$

Prove: $I_{\text{Ins}}[a, b \frown T] \Longrightarrow \underset{Y}{\exists} O_{\text{Ins}}[a, b \frown T, Y]$ that is:

$$\text{IsSorted}[b \frown T] \Longrightarrow \underset{Y}{\exists} \begin{cases} a \frown (b \frown T) \approx Y \\ \text{IsSorted}[Y] \end{cases} \quad \text{(G2)}$$

For proving (G2) we assume IsSorted[b⌢T] (H2) and

Prove: $\underset{Y}{\exists}(a \frown (b \frown T) \approx Y \bigwedge \text{IsSorted}[Y])$ (G3).

By (H2) and by Proposition "property of sort" by Modus Ponens we obtain IsSorted[T]. (H3)

From (H3) and (H1) by Modus Ponens we know: $\underset{Y}{\exists} \begin{cases} a \frown T \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (H4).

By Skolem we know: $\begin{cases} a \frown T \approx Y_0 & \text{(H4 .1)} \\ \text{IsSorted}[Y_0] & \text{(H4 .2)} \end{cases}$

We rewrite (H4.1) by definition "≈" and we know:

$$\begin{cases} a \lhd Y_0 & \text{(H4 .12)} \\ T \approx \text{dfo}[a, Y_0] & \text{(H4 .13)} \\ \text{IsSorted}[Y_0] & \text{(H4 .2)} \end{cases}$$

We have to prove (G3) that is $\underset{Y}{\exists} \begin{cases} a \frown (b \frown T) \approx Y & \text{(G3 .1)} \\ \text{IsSorted}[Y] & \text{(G3 .2)} \end{cases}$

by case distinction:

Case a≤b

Guess witness Y =$a \frown (b \frown T)$.

Thus, we obtain a≤b ⟹ Insertion[a, b⌢T]=a⌢(b⌢T). (▫▫)

Prove $\begin{cases} a \frown (b \frown T) \approx a \frown (b \frown T) & (G^* .1) \\ \text{IsSorted}[a \frown (b \frown T)] & (G^* .2) \end{cases}$

(G^*.1) is true by the definition "≈" and by the property "reflexivity in ≈".

(G^*.2) is true by the definition "IsSorted", by the fact that a≤b and by (H2).

The theory must contain the propositions:

$\text{Proposition}\left["\text{reflexivity in} \approx", \underset{X}{\forall} (X \approx X)\right]$
$\text{Proposition}\left["\text{Sorting II}", \underset{a,b,X}{\forall} ((\text{IsSorted}[b \frown X] \bigwedge (b \lhd (b \frown X)) \bigwedge (a \le b)) \Longrightarrow \text{IsSorted}[a \frown (b \frown X)])\right]$
$\text{Proposition}\left["\text{property of sort}", \underset{a,X}{\forall} (\text{IsSorted}[a \frown X] \Longrightarrow \text{IsSorted}[X])\right]$
$\text{Proposition}\left["\text{order}", \underset{a,b}{\forall} \begin{pmatrix} (\neg (a \le b)) \Longrightarrow (b < a) \\ (b < a) \Longrightarrow (b \ne a) \end{pmatrix}\right]$

Case b<a (¬(a≤b))

Guess witness Y=b⌣Y_0.

Thus, we obtain a>b ⟹ Insertion[a,b⌣T]=b⌣Y_0. (▫▫▫)

Prove: $\begin{cases} a \smile (b \smile T) \approx b \smile Y_0 & \text{(G2 .1)} \\ \text{IsSorted}[b \smile Y_0] & \text{(G2 .2)} \end{cases}$

(G2.2) is true by the Proposition["Sorting II"] because we know that b<a, $a \lhd Y_0$, IsSorted[Y_0].

Prove (G2.1): a ⌣ (b ⌣ T) ≈ b ⌣ Y_0 .

By the definition "≈" we must prove:

$\begin{cases} a \lhd (b \smile Y_0) & \text{(G5 .1)} \\ b \smile T \approx \text{dfo}[a, b \smile Y_0] & \text{(G5 .2)} \end{cases}$

In (G5.1) by the fact that a≠b (from proposition "order") and by the definition "⊲" we have to prove: a ⊲ Y_0 (G5.3) that is true by (H4.12).

We must prove (G5.2): $b \smile T \approx \text{dfo}[a, b \smile Y_0]$.

By definition "dfo" and by a≠b we must prove:

b ⌣ T ≈ b ⌣ dfo[a, Y_0] (G5.4)

By property "dfo ≈" in (G5.4) we must prove:

T ≈ dfo[a, Y_0] (G5.41) that is true by (H4.13)

By (▫), (▫▫), (▫▫▫) and by the transformation rules ($Y_0 \longrightarrow$ Insertion[a, T])
we extract the algorithm:

$$\underset{a,b,T}{\forall} \begin{cases} \textbf{Insertion}[a, \langle\rangle] = a \smile \langle\rangle \\ \textbf{Insertion}[a, b \smile T] = \begin{cases} a \smile (b \smile T) \Longleftarrow a \le b \\ b \smile \textbf{Insertion}[a, T], \text{ otherwise} \end{cases} \end{cases}$$

The algorithm "Insertion" inserts an element on the correct position into a tuple and the algorithm "$\mathcal{S}$" is the algorithm for Insertion-Sort that returns the sorted tuple.

Theory must contain the proposition:

$$\text{Proposition}\left["\text{dfo} \approx", \underset{a,T,X}{\forall} ((a \smile T \approx a \smile X) \Longrightarrow (T \approx X))\right]$$

Prove the proposition "dfo ≈":

We take a, T, X arbitrary, but fixed.

Assume: a⌣T ≈ a⌣X (H) and

Prove: T≈X (G).

By definition "≈" in (H) we know: a ⊲ a⌣X (H1) ∧ T≈dfo[a, a⌣X] (H2).

(H1) is a branch from the definition "◁".

(H2) by the definition "dfo" is: T≈X. That is our goal.

Prove the proposition "Sorting II":

$$\forall_{a,b,X} \; ((\text{IsSorted}[b \frown X] \bigwedge (b \lhd (b \frown X)) \bigwedge (a \leq b)) \Longrightarrow \text{IsSorted}[a \frown (b \frown X)])$$

We take a,b arbitrary, but fixed and we prove

$$\forall_{X} \; ((\text{IsSorted}[b \frown X] \bigwedge (b \lhd (b \frown X)) \bigwedge (a \leq b)) \Longrightarrow \text{IsSorted}[a \frown (b \frown X)]) \quad \text{(G)}$$

We prove (G) by induction over X:

Base case: X=⟨⟩

Prove: (IsSorted[⟨⟩] ∧ (b ◁ (b⌢⟨⟩)) ∧ (a ≤ b))⟹IsSorted[a⌢(b⌢⟨⟩)] (G1)

For proving (G1) we assume:

IsSorted[⟨⟩] (H1.1) ∧ (b ◁ (b⌢⟨⟩)) (H1.2) ∧ (a ≤ b) (H1.3) and

prove: IsSorted[a⌢(b⌢⟨⟩)] (G2).

This is true by (H1.3) and by definition "IsSorted".

Induction Step: X: c⌢X

Assume: (IsSorted[b⌢X] ∧ (b ◁ (b⌢X)) ∧ (a ≤ b))⟹IsSorted[a⌢(b⌢X)] (H2) and

Prove: (IsSorted[c⌢(b⌢X)] ∧ (b ◁ (c⌢(b⌢X))) ∧ (a ≤ b))

⟹IsSorted[a⌢(c⌢(b⌢X))] (G3).

For proving (G3) we assume:

IsSorted[c⌢(b⌢X)] (H3.1) ∧ (b ◁ (c⌢(b⌢X))) (H3.2) ∧ (a ≤ b) (H3.3)

Prove: IsSorted[a⌢(c⌢(b⌢X))] (G4).

By definition "IsSorted" we have to prove:

a≤c (G4.1) and IsSorted[c⌢(b⌢X)] (G4.2).

(G4.2) is true because is identical with (H3.1).

By (H3.1) and by definition "IsSorted" we know: c≤b (H3.4)

By (H1.3), by (H3.4) and by proposition "transitivity in ≤" we obtain a≤c that was our goal.

Theory must contain the proposition:

$$\text{Proposition}\Big[\text{"transitivity in } \leq\text{"}, \; \forall_{a,b,c} \; ((a \leq b) \bigwedge (b \leq c) \Longrightarrow (a \leq c))\Big]$$

3.3 Synthesis of MERGE-SORT

3.3.1 CASE I---when we know the function "merge"

We know the definitions: "≈", "dfo", "◁", "IsSorted", "concatenation", see Appendix.

Problem Specification:

$I_{\text{MSort}}[X]$: True

$$O_{\text{MSort}}[X, Y] : \begin{cases} X \approx Y \\ \text{IsSorted}[Y] \end{cases}$$

We start to **prove** the proposition:

```
Proposition["problem MergeSort",
  ∀X ∃Y P[X, Y]]
```

where

$$\text{P[X, Y]} : \begin{cases} \text{X} \approx \text{Y} \\ \text{IsSorted[Y]} \end{cases}$$

For proving the problem we use an induction principle:

$$(\text{P}[\langle\rangle] \wedge \forall_{a} \text{P}[a \frown \langle\rangle] \wedge \forall_{A,B} ((\text{P}[A] \wedge \text{P}[B]) \Longrightarrow \text{P}[A \prec B])) \Longrightarrow \forall_{A,B} P[A \prec B]$$

Case 1 : X = ⟨⟩

$$\text{Prove } \exists_{Y} P[\langle\rangle, Y] : \exists_{Y} \begin{cases} \langle\rangle \approx Y \\ \text{IsSorted}[Y] \end{cases} \text{(G1)}$$

Witness guessed : Y = ⟨⟩.

So, we obtain M[⟨⟩] = ⟨⟩. (∗)

$$\text{We must prove } \begin{cases} \langle\rangle \approx \langle\rangle & \text{(G1 .1)} \\ \text{IsSorted}[\langle\rangle] & \text{(G1 .2)} \end{cases}$$

(G1 .1) is true by the definition " ≈ ".

(G1 .2) is true by the definition " IsSorted ".

Case 2 : X : a ⌢ ⟨⟩

$$\text{Prove } \exists_{Y} P[a \frown \langle\rangle, Y] : \exists_{Y} \begin{cases} a \frown \langle\rangle \approx Y \\ \text{IsSorted}[Y] \end{cases} \text{(G2)}$$

Witness guessed : Y = a ⌢ ⟨⟩.

So, we obtain M[a ⌢ ⟨⟩] = a ⌢ ⟨⟩. (∗∗)

Prove : P[a ⌢ ⟨⟩, a ⌢ ⟨⟩] : $\begin{cases} a \frown \langle\rangle \approx a \frown \langle\rangle & \text{(G2 .1)} \\ \text{IsSorted}[a \frown \langle\rangle] & \text{(G2 .2)} \end{cases}$

(G2 .1) is true by the definition "≈" and by the proposition "reflexivity in ≈ ".

(G2 .2) is true by the definition "IsSorted".

Case 3: X : LP[X]≍RP[X]

We need to add in the knowledge base the following:

Definition["Left part of a tuple", any[a, b, X],

$$\begin{aligned} \text{LP}[\langle\rangle] &= \langle\rangle \\ \text{LP}[a \frown \langle\rangle] &= a \frown \langle\rangle \\ \text{LP}[a \frown (b \frown X)] &= a \frown \text{LP}[X] \end{aligned}$$

]

Definition["Right part of a tuple", any[a, b, X],

$$\begin{aligned} \text{RP}[\langle\rangle] &= \langle\rangle \\ \text{RP}[a \frown \langle\rangle] &= a \frown \langle\rangle \\ \text{RP}[a \frown (b \frown X)] &= b \frown \text{RP}[X] \end{aligned}$$

]

Definition["Combine the parts of a tuple", any[a, b, X, Y],

$$\begin{aligned} \text{G}[\langle\rangle, \langle\rangle] &= \langle\rangle \\ \text{G}[a \frown X, \langle\rangle] &= a \frown X \\ \text{G}[\langle\rangle, b \frown Y] &= b \frown Y \\ \text{G}[a \frown X, b \frown Y] &= \bigwedge \left\{ \begin{array}{l} a \frown \text{G}[X, b \frown Y] \Longleftarrow a \preceq b \\ b \frown \text{G}[a \frown X, Y] \Longleftarrow \text{otherwise} \end{array} \right\} \end{aligned}$$

]

Consider A=LP[X], B=RP[X].

Our problem is reduced into proving the proposition:

Proposition["reduced problem MergeSort",

$$\underset{A,B}{\forall} \left(\left(\underset{Y1}{\exists} \text{P}[A, Y1] \bigwedge \underset{Y2}{\exists} \text{P}[B, Y2] \right) \Longrightarrow \underset{Y}{\exists} \text{P}[A \asymp B, Y] \right)$$

]

Assume $\underset{Y1}{\exists}$ P[A, Y1]: $\underset{Y_1}{\exists} \begin{cases} A \approx Y_1 & \text{(H1)} \\ \text{IsSorted}[Y_1] \end{cases}$ and

$\underset{Y2}{\exists}$ P[B, Y2]: $\underset{Y_2}{\exists} \begin{cases} B \approx Y_2 & \text{(H2)} \\ \text{IsSorted}[Y_2] \end{cases}$

Prove: $\underset{Y}{\exists}$ P[A ≍ B, Y] that is $\underset{Y}{\exists} \begin{cases} (A \asymp B) \approx Y & \text{(G1).} \\ \text{IsSorted}[Y] \end{cases}$

Witness guessed is Y=G[Y_1, Y_2].

So, we obtain M[A≍B] = G[Y_1, Y_2] (***).

Prove P[A ≍ B, G[Y_1, Y_2]].

This is true by the definition "Combine the parts of a tuple".

By (*), (**), (***) and by transformation rules ($Y_1 \longrightarrow \mathrm{M}[A] \longrightarrow \mathrm{M}[\mathrm{LP}[X]]$, $Y_2 \longrightarrow \mathrm{M}[B] \longrightarrow \mathrm{M}[\mathrm{RP}[X]]$, A≍B=X) we obtain the algorithm:

$$\forall_{a,X} \begin{cases} \mathbf{M[\langle\rangle] = \langle\rangle} \\ \mathbf{M}[a \smile \langle\rangle] = a \smile \langle\rangle \\ \mathbf{M[}X\mathbf{] = }G\mathbf{[M[LP[}X\mathbf{]], M[RP[}X\mathbf{]]]} \end{cases}$$

3.3.2 CASE II--when we don't know the function "merge" and we synthetise it

Reduced Problem 1:

Specification:

Given Y_1 and Y_2 sorted, find Y such that:

$$\forall_{\substack{\mathrm{IsSorted}[Y_1] \\ \mathrm{IsSorted}[Y_2]}} \exists_{Y} \mathrm{P}[Y_1 \asymp Y_2,\ Y] : \exists_{Y} \begin{cases} (Y_1 \asymp Y_2) \approx Y \\ \mathrm{IsSorted}[Y] \end{cases}$$

We start to **prove** the proposition:

$$\mathrm{Proposition}\left["\text{reduced problem Merge }",\ \forall_{Y_1,Y_2} \exists_{Y} ((Y_1 \asymp Y_2) \approx Y \bigwedge \mathrm{IsSorted}[Y])\right]$$

using the induction principle:

$$\left(P[\langle\rangle, \langle\rangle] \bigwedge \forall_{a,X} (P[X, \langle\rangle] \Longrightarrow P[a \smile X, \langle\rangle]) \bigwedge \right.$$

$$\forall_{b,Y} (P[\langle\rangle, Y] \Longrightarrow P[\langle\rangle, b \smile Y]) \bigwedge$$

$$\left. \forall_{a,b,X,Y} (P[X, Y] \Longrightarrow P[a \smile X, b \smile Y]) \right)$$

$$\Longrightarrow \forall_{X,Y} P[X, Y]$$

We have 4 cases:

1) $Y_1 : \langle\rangle$ and $Y_2 : \langle\rangle$
2) $Y_1 : a \smile Y_1$ and $Y_2 = \langle\rangle$
3) $Y_1 : \langle\rangle$ and $Y_2 : b \smile Y_2$
4) $Y_1 : a \smile Y_1$ and $Y_2 : b \smile Y_2$

Case 1) $Y_1 : \langle\rangle$ *and* $Y_2 : \langle\rangle$

Prove: $\exists_Y P[\langle\rangle \asymp \langle\rangle, Y] : \exists_Y \begin{cases} (\langle\rangle \asymp \langle\rangle) \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (G1)

Witness guessed is Y=⟨⟩.

Thus, we obtain G[⟨⟩,⟨⟩]=⟨⟩. (■)

Prove P[⟨⟩ ≍ ⟨⟩, ⟨⟩]: $\begin{cases} (\langle\rangle \asymp \langle\rangle) \approx \langle\rangle & \text{(G1 .1)} \\ \text{IsSorted}[\langle\rangle] & \text{(G1 .2)} \end{cases}$

(G1.1) by the definition "concatenation" is ⟨⟩≈⟨⟩ (G1.12).

This is true by the definition "≈".

(G1.2) is true by the definition "IsSorted".

Case 2) $Y_1 : a \frown Y_1$ *and* $Y_2 : \langle\rangle$

We know that IsSorted[$a \frown Y_1$].

Prove: $\exists_Y P[(a \frown Y_1) \asymp \langle\rangle, Y] : \exists_Y \begin{cases} ((a \frown Y_1) \asymp \langle\rangle) \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (G2)

Witness guessed is Y=a$\frown Y_1$.

Thus, we obtain G[(a$\frown Y_1$), ⟨⟩] = a$\frown Y_1$. (■■)

Prove P[(a$\frown Y_1$) ≍ ⟨⟩, a$\frown Y_1$]: $\begin{cases} ((a \frown Y_1) \asymp \langle\rangle) \approx a \frown Y_1 & \text{(G2 .1)} \\ \text{IsSorted}[a \frown Y_1] & \text{(G2 .2)} \end{cases}$

By definition "≍" we must prove:

$\begin{cases} (a \frown Y_1) \approx a \frown Y_1 & \text{(G2 .11)} \\ \text{IsSorted}[a \frown Y_1] & \text{(G2 .2)} \end{cases}$

(G2.11) is true by the definition "≈" and by proposition "reflexivity in ≈".

Prove (G2.2): IsSorted[a $\frown$ Y_1] this is true because it was our assumption.

Theory must contain the proposition:

Proposition["reflexivity in ≈", $\forall_X (X \approx X)$]

Case 3) $Y_1 : \langle\rangle$ *and* $Y_2 : b \frown Y_2$

We know that IsSorted[$b \frown Y_2$].

Prove : $\exists_Y P[\langle\rangle \asymp (b \frown Y_2), Y] : \exists_Y \begin{cases} (\langle\rangle \asymp (b \frown Y_2)) \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (G3)

Witness guessed is Y = (b $\frown$ Y_2).

Thus, we obtain $G[\langle\rangle, b \frown Y_2] = b \frown Y_2$. (■■■)

Prove $P[\langle\rangle \asymp (b \frown Y_2), b \frown Y_2]$: $\begin{cases} (\langle\rangle \asymp (b \frown Y_2)) \approx b \frown Y_2 & \text{(G3 .1)} \\ \text{IsSorted}[b \frown Y_2] & \text{(G3 .2)} \end{cases}$

By definition "$\asymp$" we must prove : $\begin{cases} (b \frown Y_2) \approx (b \frown Y_2) & \text{(G3 .11)} \\ \text{IsSorted}[(b \frown Y_2)] & \text{(G3 .2)} \end{cases}$

(G3 .11) is true by the definition " $\approx$" and by proposition " reflexivity in $\approx$ ".

(G3 .2) is true because it was our assumption.

Case 4) $Y_1 : a \frown Y_1$ *and* $Y_2 : b \frown Y_2$

We know that IsSorted$[a \frown Y_1]$, IsSorted$[b \frown Y_2]$

Our problem is reduced to prove the proposition:

```
Proposition["reduced problem Merge 2.1", any[a, b, Y1, Y2],
  (G[Y1, b ⌢ Y2] ⋀ G[a ⌢ Y1, Y2]) ⟹ G[a ⌢ Y1, b ⌢ Y2]]
```

Where $G[Y_1, b \frown Y_2]$ is $\exists_{Z_1} \begin{cases} Y_1 \asymp (b \frown Y_2) \approx Z_1 \\ \text{IsSorted}[Z_1] \end{cases}$

$G[a \frown Y_1, Y_2]$ is $\exists_{Z_2} \begin{cases} (a \frown Y_1) \asymp Y_2 \approx Z_2 \\ \text{IsSorted}[Z_2] \end{cases}$

$G[a \frown Y_1, b \frown Y_2]$ is $\exists_{Z} \begin{cases} (a \frown Y_1) \asymp (b \frown Y_2) \approx Z \\ \text{IsSorted}[Z] \end{cases}$

We start to prove the proposition "reduced problem Merge 2.1".

We assume: $\exists_{Z_1} \begin{cases} Y_1 \asymp (b \frown Y_2) \approx Z_1 \\ \text{IsSorted}[Z_1] \end{cases}$ (H1) and $\exists_{Z_2} \begin{cases} (a \frown Y_1) \asymp Y_2 \approx Z_2 \\ \text{IsSorted}[Z_2] \end{cases}$ (H2)

By Skolem we know:

$\begin{cases} Y_1 \asymp (b \frown Y_2) \approx Z_1 & \text{(H1 .1)} \\ \text{IsSorted}[Z_1] & \text{(H1 .2)} \end{cases}$ and $\begin{cases} (a \frown Y_1) \asymp Y_2 \approx Z_2 & \text{(H2 .1)} \\ \text{IsSorted}[Z_2] & \text{(H2 .2)} \end{cases}$

We prove: $\exists_{Z} \begin{cases} (a \frown Y_1) \asymp (b \frown Y_2) \approx Z \\ \text{IsSorted}[Z] \end{cases}$ (G)

Witness guessed: $Z = a \frown Z_1$

We have to prove: $\begin{cases} (a \frown Y_1) \asymp (b \frown Y_2) \approx a \frown Z_1 & \text{(G1 .1)} \\ \text{IsSorted}[a \frown Z_1] & \text{(G1 .2)} \end{cases}$

By (H1.1) we must prove: $\begin{cases} (a \frown Y_1) \asymp (b \frown Y_2) \approx a \frown (Y_1 \asymp (b \frown Y_2)) & \text{(G1 .11)} \\ \text{IsSorted}[a \frown Z_1] & \text{(G1 .2)} \end{cases}$

Prove (G1.11): $(a \frown Y_1) \asymp (b \frown Y_2) \approx a \frown (Y_1 \asymp (b \frown Y_2))$

By the definition "concatenation" we must prove:

$a \frown (Y_1 \asymp (b \frown Y_2)) \approx a \frown (Y_1 \asymp (b \frown Y_2))$ (G1.12)

This is true by the proposition "dfo ≈" and by the proposition "reflexivity in ≈".

Prove (G1.2): IsSorted[$a \frown Z_1$]

By proposition "element in concatenation" and by (H1.1) we know that $b \lhd Z_1$ (H1.3).

The theory must contain the propositions:

$$\text{Proposition}\left["\text{dfo} \approx",\ \forall_{a,T,X}\ ((a \frown T \approx a \frown X) \Longrightarrow (T \approx X))\right]$$

$$\text{Proposition}\left["\text{element in concatenation}",\ \forall_{a,A,B,C}\ \begin{pmatrix}((A \asymp (a \frown B)) \approx C) \Longrightarrow (a \lhd C)\\ (((a \frown A) \asymp B) \approx C) \Longrightarrow (a \lhd C)\end{pmatrix}\right]$$

By (H1.2), (H1.3) iff a≤b then we can use the proposition "Sorting II" and we obtain IsSorted[$a \frown Z_1$] that was our goal.

So, a≤b ⟹ *G*[$a \frown Y_1, b \frown Y_2$] = $a \frown Z_1$ (&&&)

The theory must contain the proposition "Sorting II":

$$\text{Proposition}\left["\text{Sorting II}",\ \forall_{a,b,X}\ ((\text{IsSorted}[b \frown X] \bigwedge (b \lhd (b \frown X)) \bigwedge (a \le b)) \Longrightarrow \text{IsSorted}[a \frown (b \frown X)])\right]$$

Case a>b

We have to prove $\exists_Z \begin{cases}(a \frown Y_1) \asymp (b \frown Y_2) \approx Z\\ \text{IsSorted}[Z]\end{cases}$ (G)

Witness guessed: $Z = b \frown Z_2$

We have to prove: $\begin{cases}(a \frown Y_1) \asymp (b \frown Y_2) \approx b \frown Z_2 & \text{(G2 .1)}\\ \text{IsSorted}[b \frown Z_2] & \text{(G2 .2)}\end{cases}$

Prove (G2.1): $(a \frown Y_1) \asymp (b \frown Y_2) \approx b \frown Z_2$

By (H2.1) we must prove: $(a \frown Y_1) \asymp (b \frown Y_2) \approx b \frown ((a \frown Y_1) \asymp Y_2)$ (G2.11)

By the definition "concatenation" we must prove:

$a \frown (Y_1 \asymp (b \frown Y_2)) \approx b \frown ((a \frown Y_1) \asymp Y_2)$ (G2.12)

By the definition "≈" we must prove:

$$\begin{cases}a \lhd b \frown ((a \frown Y_1) \asymp Y_2) & \text{(G3)}\\ (Y_1 \asymp (b \frown Y_2)) \approx \text{dfo}[a, b \frown ((a \frown Y_1) \asymp Y_2)] & \text{(G4)}\end{cases}$$

Prove (G3): $a \lhd b \frown ((a \frown Y_1) \asymp Y_2)$

By the proposition "order" we know that a≠b (H1.4).

By (H1.4) and by the definition "⊲" we must prove:

$a \lhd ((a \frown Y_1) \asymp Y_2)$ (G3.1).

By the definition "concatenation" we must prove:

$a \lhd a \frown (Y_1 \asymp Y_2)$ (G3.2)

This is true by the definition "$\lhd$".

The theory must contain the proposition:

$$\text{Proposition}\left["order",\ \forall_{a,b}\begin{pmatrix}(\neg(a \le b)) \Longrightarrow (b < a)\\ (b < a) \Longrightarrow (b \ne a)\end{pmatrix}\right]$$

Prove (G4): $(Y_1 \asymp (b \frown Y_2)) \approx \text{dfo}[a, b \frown ((a \frown Y_1) \asymp Y_2)]$

By (H1.4) and by the definition "dfo" we must prove:

$(Y_1 \asymp (b \frown Y_2)) \approx \text{dfo}[a, ((a \frown Y_1) \asymp Y_2)]$ (G4.1)

By the definition "concatenation" we must prove:

$(Y_1 \asymp (b \frown Y_2)) \approx b \frown \text{dfo}[a, a \frown (Y_1 \asymp Y_2)]$ (G4.2)

By the definition "dfo" we must prove:

$(Y_1 \asymp (b \frown Y_2)) \approx b \frown (Y_1 \asymp Y_2)$ (G4.3)

This is true by the proposition "the same elements in concatenation".

Theory must contain the proposition:

$$\text{Proposition}\left["the\ same\ elements\ in\ concatenation",\ \forall_{a,X,Y}(X \asymp (a \frown Y) \approx a \frown (X \asymp Y))\right]$$

Prove (G2.2): IsSorted$[b \frown Z_2]$

By (H2.1) and by the proposition "elements in concatenation" we know that b<a (H2.3).

By (H2.3), (H2.2) and by the proposition "Sorting II" we obtain that IsSorted[b $\frown$ Z_2] is true.

So, we obtain that $\mathbf{a>b \Longrightarrow G[a \frown Y_1, b \frown Y_2] = b \frown Z_2}$ (&&&&)

By (■), (■■), (■■■), (&&&), (&&&&) and by the transformation rules ($Z_1 \longrightarrow G[Y_1, b \frown Y_2], Z_2 \longrightarrow G[a \frown Y_1, Y_2]$) we obtain the **algorithm for G:**

$$\forall_{a,b,A,B}\begin{cases}\mathbf{G[\langle\rangle, \langle\rangle] = \langle\rangle}\\ \mathbf{G[(a \frown Y_1), \langle\rangle] = a \frown Y_1}\\ \mathbf{G[\langle\rangle, b \frown Y_2] = b \frown Y_2}\\ \mathbf{G[a \frown Y_1, b \frown Y_2]} = \begin{cases}\mathbf{a \frown G[Y_1, b \frown Y_2] \Longleftarrow a \le b}\\ \mathbf{b \frown G[a \frown Y_1, Y_2]\ otherwise}\end{cases}\end{cases}$$

So, by the transformation rules ($Y_1 \longrightarrow$M[LP[X]], $Y_2 \longrightarrow$M[RP[X]]) we obtain the final **algorithm for sorting:**

$$\forall_{a,X} \begin{cases} \mathbf{M}[\langle\rangle] = \langle\rangle \\ \mathbf{M}[\boldsymbol{a} \smile \langle\rangle] = \boldsymbol{a} \smile \langle\rangle \\ \mathbf{M}[\boldsymbol{X}] = \boldsymbol{G}[\mathbf{M}[\mathbf{LP}[\boldsymbol{X}]], \mathbf{M}[\mathbf{RP}[\boldsymbol{X}]]] \end{cases}$$

The algorithm for G corresponds to "Merge" that combines the parts of a tuple and the algorithm M is the algorithm for "Merge-Sort".

Chapter 4

Synthesis of Sorting Algorithms on Tuples-the case when we find the witness

4.1 Description of the method

In this section we give a general description of our approach.

We start from the specification of the problem (input and output condition). Given the *Problem Specification:*

Input: $I_F[X]$

Output: $O_F[X, Y]$

find the definition of F such that $\underset{X:I_F}{\forall} O_F[X, F[X]]$. Note that in our formalism we use square brackets as in $f[x]$ for function application and for predicate application, instead of the usual round parantheses as in $f(x)$. Also, the quantified formula above is a notation for: $(\forall X)(I_F[X] \Longrightarrow O_F[X, F[X]])$.

We prove $\underset{X:I_F}{\forall} \underset{Y}{\exists} O_F[X, Y]$, either directly or using an induction principle (the choice of the proof style and of the induction principle is not automatic).

Direct proof. We take X_0 arbitrary but fixed (a new constant), we assume $I_F[X_0]$ and by proof we find a witness $T[X_0]$ (a term depending on X_0) such that $O_F[X_0, T[X_0]]$. In this case, the algorithm is $F[X] = T[X]$.

Inductive proof. Let us consider for illustration the head-tail inductive definition of tuples. In our notation, $\langle\rangle$ represents the empty tuple, $a \frown \langle\rangle$ is a tuple having a single element a, and $a \frown X$ is the tuple having head a and tail X. Complementary to this inductive definition, we consider that the functions Head and Tail are in the knowledge base. Moreover, in order for this induction scheme to be appropriate for our algorithm, one must have the properties: $I_F[\langle\rangle]$, and $\underset{a}{\forall}\underset{X}{\forall} I_F[a \frown X] \Rightarrow I_F[X]$. (That is: the input condition must hold for the base case and for the inductive decomposition of the argument.)

Base case: We prove $\underset{Y}{\exists} O_F[\langle\rangle, Y]$. If the proof succeeds to find a ground term C such

that $O_F[\langle\rangle, C]$, then we know that $F[\langle\rangle] = C$.

Induction step: For arbitrary but fixed a and X_0 (satisfying I_F), we assume $\underset{Y}{\exists} O_F[X_0, Y]$ and we prove $\underset{Y}{\exists} O_F[a \frown X_0, Y]$. We Skolemize the assumption by introducing a new constant Y_0 for the existential Y. If the proof succeeds to find a witness $T[a, X_0, Y_0]$ (term depending on a, X_0, and Y_0) such that $O_F[a \frown X_0, T[a, X_0, Y_0]]$, then we know that $F[a \frown X] = T[a, X, F[X]$.

Algorithm extraction: Finally the algorithm is expressed as:

$$F[X] = \begin{cases} C, & \text{if} X = \langle\rangle \\ T[Head[X], Tail[X], F[Tail[X]], & \text{if} X \neq \langle\rangle \end{cases}$$

Cascading. Assume that in the induction step above the prover cannot find a witness. The we can create the following new problem: "Given a, X, Y such that $I_F[X]$ and $O[X, Y]$, find a Z such that $O_F[a \frown X, Z]$." That is, if we cannot find an appropriate term, we try to synthesize (by a possibly different induction principle) a new function $G[a, X, Y]$ which does the job. Syntactically this new problem looks more complicated, however it is logically simpler, because the arguments fulfill more conditions, while the output requirement is the same as before. As we will demonstrate below, this cascading principle can be repeated, and the problem will be reduced into smaller and smaller problems until the necessary functions are already in the knowledge base.

During the proof we discover propositions that we need for the proof in order to succeed. We prove these propositions and after that introduce them into the knowledge and use them in the proof. We will not go into details in this paper for the process of inventing propositions. We just remark that by adding propositions to the knowledge base we explore the theory. Thus the process of proving is paralleled to the process of exploring (building) the theory, and the case study presented here increases our experience in developing a reasonable theory of tuples for further practical applications.

4.2 Synthesis of INSERTION SORT (2)

4.2.1 Case when we know the function "Insertion"

We know the definitions "≈", "dfo", "◅", "IsSorted", see Appendix.

The theory contains also the definition "Insertion":

```
Definition["Insertion", any[a, b, X],
                    Insertion[a, ⟨⟩] = ⟨a⟩
      (a ≤ b) ⟹ (Insertion[a, b ⌢ X] = a ⌢ (b ⌢ X))
      (a > b) ⟹ (Insertion[a, b ⌢ X] = b ⌢ Insertion[a, X])]
```

Specification:

$$I_{\text{InsertSort}}[X]: \text{True}$$

$$O_{\text{InsertSort}}[X, Y]: \begin{cases} X \approx Y \\ \text{IsSorted}[Y] \end{cases}$$

```
Proposition["problem InsertionSort",
  ∀ ∃ P[X, Y]]
  X Y
```

where

```
P[X, Y]: { X ≈ Y
         { IsSorted[Y]
```

We prove the proposition "problem InsertionSort" by induction over X.

We use the induction principle:

$$\left(P[\langle\rangle] \bigwedge \underset{a,X}{\forall} (P[X] \Longrightarrow P[a \frown X])\right) \Longrightarrow \underset{X}{\forall} P[X]$$

The induction principle is in second order logic, but by instantiation it becomes first order.

Base case: X=⟨⟩

$$\text{Prove } \underset{Y}{\exists} P[\langle\rangle, Y]: \underset{Y}{\exists} \begin{cases} \langle\rangle \approx Y \\ \text{IsSorted}[Y] \end{cases} \quad \text{(G1)}$$

$$\text{Find witness } Y^? \text{ such that } \begin{cases} \langle\rangle \approx Y^? & \text{(G1 .1)} \\ \text{IsSorted}[Y^?] & \text{(G1 .2)} \end{cases}$$

(G1.1) By matching ⟨⟩≈⟨⟩ from the definition "≈" with $\langle\rangle \approx Y^?$ we find *witness* $Y^? = \langle\rangle$.

We *check for (G1.2)*: IsSorted[⟨⟩] is true by definition "IsSorted".
So, we obtain $\mathcal{S}[\langle\rangle]=\langle\rangle$. (*)

Induction Step:

Assume: $P[X_0, Y_0]: \begin{cases} X_0 \approx Y_0 & \text{(H1 .1)} \\ \text{IsSorted}[Y_0] & \text{(H1 .2)} \end{cases}$ and

Prove: $\underset{Y}{\exists} P[a \frown X_0, Y]: \begin{cases} a \frown X_0 \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (G)

Find witness $Y^?$ such that: $\begin{cases} a \frown X_0 \approx Y^? & \text{(G2)} \\ \text{IsSorted}[Y^?] & \text{(G2 .3)} \end{cases}$

By definition "≈" we obtain: $\begin{cases} a \lhd Y^? & \text{(G2 .1)} \\ X_0 \approx \text{dfo}[a, Y^?] & \text{(G2 .2)} \\ \text{IsSorted}[Y^?] & \text{(G2 .3)} \end{cases}$

Theory must contain the propositions:

Proposition["IsElem in Insertion", $\underset{a,X}{\forall}$ (a ◁ Insertion[a, X])]

Proposition["The same elements in Insertion", $\underset{a,X,Y}{\forall}$ ((X ≈ Y) ⟹ (X ≈ dfo[a, Insertion[a, Y]]))]

Proposition["Sorting using Insertion", $\underset{a,X}{\forall}$ (IsSorted[X] ⟹ IsSorted[Insertion[a, X]])]

We instantiate proposition "IsElem in Insertion" and we obtain:
a ◁ Insertion[a, X_0] (H2.1).

By matching (H2.1) with (G2.1) we find *witness* $Y^? = Insertion[a, Y_0]$.

Check for (G2.2):

$X_0 \approx$ dfo[a, Insertion[a, Y_0]] .

We instantiate "The same elements in Insertion" and we have:
$(X_0 \approx Y_0) \Longrightarrow (X_0 \approx$ dfo[a, Insertion[a, Y_0]]) (H2.2)

By (H2.2), (H1.1), by Modus Ponens we know:
$X_0 \approx$ dfo[a, Insertion[a, Y_0]] (H2.3) that is the same with (G2.2).

Check for (G2.3):

IsSorted[Insertion[a, Y_0]]

Instantiate "Sorting using Insertion" and we know:
IsSorted[Y_0] ⟹ IsSorted[Insertion[a, Y_0]] (H2.4)

By (H2.4), (H1.2), by Modus Ponens we obtain:

IsSorted[Insertion[a, Y_0]] that is the same with (G2.3).

Because we know the definition "Insertion" we do matching and obtain:

$\mathcal{S}$[a⌣X_0] = Insertion[a, Y_0] (**)

By (*), (**) and by the transformation rules $X_0 \longrightarrow X$, $Y_0 \longrightarrow \mathcal{S}[X]$ we extract the **algorithm:**

$$\forall_{a,X} \begin{cases} \mathcal{S}[\langle\rangle] = \langle\rangle \\ \mathcal{S}[a \smile X] = \text{Insertion}[a, \mathcal{S}[X]] \end{cases}$$

4.2.2 Case when we do not know the function "Insertion" and we synthetize it

We know the definitions: "≈", "dfo", "◁", "IsSorted", see Appendix.

We write X as an element a added to a tuple T. The input condition is extended by adding that T is sorted and the requirement for the output remains the same.

Reduced Problem 1:

Specification: (X: a,T)

$I_{\text{Ins}}[a, T]$: IsSorted[T]

$$O_{\text{Ins}}[a, T, Y] : \begin{cases} a \smile T \approx Y \\ \text{IsSorted}[Y] \end{cases}$$

Generate the conjecture:

$$\text{Proposition}\left["\text{reduced problem Insertion}", \forall_{a,T}\left(\text{IsSorted}[T] \Longrightarrow \exists_{Y}((a \smile T \approx Y) \bigwedge \text{IsSorted}[Y])\right)\right]$$

Prove the proposition "reduced problem Insertion" by induction over T:

We use the Induction Principle Head/Tail (Decomposition):

$$\left(P[\langle\rangle] \bigwedge \forall_{b,T} (P[T] \Longrightarrow P[b \smile T])\right) \Longrightarrow \forall_{T} P[T]$$

We take *a* arbitrary, but fixed and we prove:

Base case: T=⟨⟩

$$\text{Prove } \exists_{Y} \begin{cases} a \smile \langle\rangle \approx Y \\ \text{IsSorted}[Y] \end{cases} \quad \text{(G*3)}$$

Find witness $Y^?$ such that $\begin{cases} a \frown \langle\rangle \approx Y^? & \text{(G3 .1)} \\ \text{IsSorted}[Y^?] & \text{(G3 .2)} \end{cases}$

Theory must contain the proposition:

$\text{Proposition}\left["\text{reflexivity in } \approx",\ \underset{X}{\forall}\,(X \approx X)\right]$

We use the proposition "reflexivity in ≈" and by matching with (G3.1) we find witness $Y^? = a \frown \langle\rangle$.

Check for (G3.2): IsSorted[a⌢⟨⟩]. This is true by the definition "IsSorted".

So, we obtain Ins[a, ⟨⟩] = a ⌢ ⟨⟩. (*)

Induction Step: T: b⌢T

Assume: $\text{IsSorted}[T] \Longrightarrow \underset{Y}{\exists} \begin{cases} a \frown T \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (H*4)

Prove: $\text{IsSorted}[b \frown T] \Longrightarrow \underset{Y}{\exists} \begin{pmatrix} a \frown (b \frown T) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G*4)

For proving (G*4) we assume IsSorted[b⌢T] (H4) and

prove: $\underset{Y}{\exists} \begin{pmatrix} a \frown (b \frown T) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G4)

By (H4) and "property of sort" we obtain IsSorted[T] (H5).

From (H5), (H*4), by Modus Ponens we know: $\underset{Y}{\exists} \begin{cases} a \frown T \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (H6)

In (H6), by Skolem we obtain:

$\begin{cases} a \frown T \approx Y_0 & \text{(H6 .1)} \\ \text{IsSorted}[Y_0] & \text{(H6 .2)} \end{cases}$

By the definition of "≈" we rewrite (H6.1) into:

$\begin{cases} a \lhd Y_0 & \text{(H7 .1)} \\ T \approx \text{dfo}[a, Y_0] & \text{(H7 .2)} \\ \text{IsSorted}[Y_0] & \text{(H6 .2)} \end{cases}$

Prove (G4):

Find witness $Y^?$ such that $\begin{cases} a \frown (b \frown T) \approx Y^? & \text{(G4 .1)} \\ \text{IsSorted}[Y^?] & \text{(G4 .2)} \end{cases}$

Alternative I:

Using proposition "reflexivity in ≈" we do matching in (G4.1) and obtain witness:

$Y^? = a \frown (b \frown T)$.

Check in (G4.2): IsSorted[a⌣(b⌣T)].
By definition "IsSorted" we have to prove:
(a≤b) ∧ IsSorted[b⌣T].
IsSorted[b⌣T] is true by H4.
So, (a≤b)$\Longrightarrow Y^{?} = a \smile (b \smile T)$ and we obtain:
Ins[a,b⌣T]=a⌣(b⌣T) iff a≤b. (**)

Alternative II:

By IsSorted[Y_0] (H6.2), a $\lhd Y_0$ (H7.1) and by ¬(a≤b) (from proposition "order") and if we consider that a is the smallest element from Y_0 then we can use the property "Sorting II" and we obtain that IsSorted[$b \smile Y_0$] (H5).

We do matching of (H5) with (G4.2) and we obtain witness $Y^{?} = b \smile Y_0$.

Check in (G4.1): a⌣(b⌣T)≈$b \smile Y_0$.

By the definition "≈" we rewrite this into:

a $\lhd b \smile Y_0$ (G4.12)

$b \smile T \approx$ dfo[a, $b \smile Y_0$] (G4.13)

By the proposition "order" we know that a≠b (H5.1)

By the definition "dfo" and by (H5.1) we rewrite (G4.13) into:

$b \smile T \approx b \smile$ dfo[a, Y_0] (G4.14)

So, we have to prove:

$$\begin{cases} a \lhd b \smile Y_0 & \text{(G4 .12)} \\ b \smile T \approx b \smile \text{dfo}[a, Y_0] & \text{(G4 .14)} \end{cases}$$

Theory must contain the proposition:

$$\text{Proposition}\left["\text{order}", \forall_{a,b} \begin{array}{c} (\neg(a \le b)) \Longrightarrow (a > b) \\ a > b \Longrightarrow a \ne b \end{array}\right]$$

$$\text{Proposition}\left["\text{dfo} \approx", \forall_{a,T,X} ((a \smile T \approx a \smile X) \Longrightarrow (T \approx X))\right]$$

Prove (G4.14) By proposition "dfo ≈" we have to prove:

$T \approx$ dfo[a, Y_0] that is true by (H7.2).

Prove (G4.12): a $\lhd b \smile Y_0$

By definition "⊲" and by (H5.1) we obtain a $\lhd Y_0$ that is true by (H7.1).

So, our witness is correct: $Y^{?} = b \frown Y_0$.

So, we obtain Ins[a,b⌢T]=b⌢Y_0 iff a>b. (***)

Theory must contain the propositions:

$$\text{Proposition}\left["\text{Sorting II}",\ \forall_{a,b,X}\ ((\text{IsSorted}[b \frown X] \wedge (b \lhd (b \frown X)) \wedge (a \le b)) \Longrightarrow \text{IsSorted}[a \frown (b \frown X)])\right]$$

$$\text{Proposition}\left["\text{property of sort}",\ \forall_{a,X}\ (\text{IsSorted}[a \frown X] \Longrightarrow \text{IsSorted}[X])\right]$$

From (*), (**), (***) and by the transformation rules $Y_0 \longrightarrow$ Ins[a, T], T⟶X

we obtain the algorithm:

$$\forall_{a,b,X}\left(\begin{array}{c} \boldsymbol{Ins[a,\ \langle\rangle] = a \frown \langle\rangle} \\ \boldsymbol{Ins[a,\ b \frown X] = \begin{cases} a \frown (b \frown X), & a \le b \\ b \frown Ins[a,\ X] & , a > b \end{cases}} \end{array}\right)$$

4.3 Synthesis of MERGE-SORT(2)

We know the definitions: "≈", "dfo", "⊲", "IsSorted", "concatenation", see Appendix.

Problem *Specification:*

$I_{\text{MSort}}[X]$: True

$$O_{\text{MSort}}[X, Y]: \begin{cases} X \approx Y \\ \text{IsSorted}[Y] \end{cases}$$

We start to prove the proposition:

$$\text{Proposition}\left["\text{problem MergeSort(2)}",\ \forall_X \exists_Y \begin{pmatrix} X \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}\right]$$

It is sufficient to prove the proposition "problem MergeSort(2)" because we do not take into account the type of the objects. We consider that all the objects are tuples and the fact that we do not check the type of the objects does not influence our proof.

For proving we use the induction principle:

$$(P[\langle\rangle] \wedge \forall_a P[a \frown \langle\rangle] \wedge$$

$$\forall_{A,B}\ ((P[A] \wedge P[B]) \Longrightarrow P[A \asymp B]))$$

$$\Longrightarrow \forall_{A,B} P[A \asymp B]$$

The induction principle corresponds to the "divide-and-conquer" algorithm design scheme. We assume that this comes together with two functions LP (LeftParts) and RP (RightParts) which split a tuple into two disjoint tuples. Of course there can be many such functions, but the choice is not important for this synthesis process. It may be important for the efficiency in certain environments.

Case 1: X=⟨⟩

Prove $\exists_{Y}\left(\begin{array}{c}\langle\rangle \approx Y\\ \text{IsSorted}[Y]\end{array}\right)$ (G1)

Find witness $Y^{?}$ such that $\left\{\begin{array}{ll}\langle\rangle \approx Y^{?} & \text{(G1 .1)}\\ \text{IsSorted}\left[Y^{?}\right] & \text{(G1 .2)}\end{array}\right.$

By matching ⟨⟩≈⟨⟩ (from the definition "≈") with $\langle\rangle \approx \mathrm{Y}^{?}$ we obtain $\mathrm{Y}^{?} = \langle\rangle$.

So, we obtain **M[⟨⟩]=⟨⟩. (*)**

Check for (G1.2): IsSorted[⟨⟩]. This is true by the definition of "IsSorted".

Case 2: X=a⌣⟨⟩

Prove $\exists_{Y}\left(\begin{array}{c}a \smile \langle\rangle \approx Y\\ \text{IsSorted}[Y]\end{array}\right)$ (G2)

Find witness $Y^{?}$ such that $\left\{\begin{array}{ll}a \smile \langle\rangle \approx Y^{?} & \text{(G2 .1)}\\ \text{IsSorted}\left[Y^{?}\right] & \text{(G2 .2)}\end{array}\right.$

We do matching with proposition "reflexivity in ≈" and we find witness $Y^{?} = a \smile \langle\rangle$.

Check for (G2.2): IsSorted[a⌣⟨⟩]. This is true by definition "IsSorted".

So, we obtain **M[a⌣⟨⟩]=a⌣⟨⟩. (**)**

Case 3: X=A≍B

Prove: $\left(\exists_{Y_1}\left(\begin{array}{c}A \approx Y_1\\ \text{IsSorted}[Y_1]\end{array}\right)\bigwedge\exists_{Y_2}\left(\begin{array}{c}B \approx Y_2\\ \text{IsSorted}[Y_2]\end{array}\right)\right)\Longrightarrow\exists_{Y}\left(\begin{array}{c}(A \asymp B) \approx Y\\ \text{IsSorted}[Y]\end{array}\right)$

For proving this we assume: $\exists_{\mathrm{Y}_1}\left(\begin{array}{c}\mathrm{A} \approx \mathrm{Y}_1\\ \text{IsSorted}[\mathrm{Y}_1]\end{array}\right)\bigwedge\exists_{\mathrm{Y}_2}\left(\begin{array}{c}\mathrm{B} \approx \mathrm{Y}_2\\ \text{IsSorted}[\mathrm{Y}_2]\end{array}\right)$

By Skolem,

Assume: $\left\{\begin{array}{ll}A \approx Y_1 & \text{(H1 .1)}\\ \text{IsSorted}[Y_1] & \text{(H1 .2)}\end{array}\right.$ and $\left\{\begin{array}{ll}B \approx Y_2 & \text{(H1 .3)}\\ \text{IsSorted}[Y_2] & \text{(H1 .4)}\end{array}\right.$ and

Prove: $\exists_{Y}\left(\begin{array}{c}(A \asymp B) \approx Y\\ \text{IsSorted}[Y]\end{array}\right)$ (G3)

Find witness $Y^{?}$ such that $\left\{\begin{array}{ll}(A \asymp B) \approx Y^{?} & \text{(G3 .1)}\\ \text{IsSorted}\left[Y^{?}\right] & \text{(G3 .2)}\end{array}\right.$

From (H1.1), (H1.2), (H1.3), (H1.4) we induce that our witness is a function depending

on A, B, Y_1, Y_2.

The function F that combines two sorted tuples may or may not be in the knowledge base.

4.3.1 Case when we know the function F

When we have the definition "Combine the parts of a tuple":

$$\text{Definition}\Big["\text{Combine the parts of a tuple}",\ \text{any}[a, b, X, Y],$$
$$\begin{Bmatrix} F[\langle\rangle, \langle\rangle] = \langle\rangle \\ F[a \frown X, \langle\rangle] = a \frown X \\ F[\langle\rangle, b \frown Y] = b \frown Y \end{Bmatrix}$$
$$F[a \frown X, b \frown Y] = \bigwedge \left\{ \begin{matrix} a \frown F[X, b \frown Y] \Longleftarrow a \preceq b \\ b \frown F[a \frown X, Y] \Longleftarrow \text{otherwise} \end{matrix} \right\}$$

we match the definition and obtain our witness $Y^? = F[Y_1, Y_2]$ which satisfies the conditions that $F[Y_1, Y_2] \approx Y_1 \asymp Y_2$ and IsSorted[$F[Y_1, Y_2]$].

It is reasonable that if the definition of F is in the knowledge base, then these properties are also on the knowledge base, because they are actually the reason why such a function is present.

Thus, we obtain M[A $\asymp$ B]=$F[Y_1, Y_2]$ (***)

So, by (*), (**), (***) and by the transformation rules A$\asymp$B$\longrightarrow$X, A$\longrightarrow$LP[X], B$\longrightarrow$ RP[X], $Y_1 \longrightarrow$M[LP[X]], $Y_2 \longrightarrow$M[RP[X]] we obtain the algorithm:

$$\underset{a,X}{\forall} \left(\begin{matrix} \boldsymbol{M[\langle\rangle] = \langle\rangle} \\ \boldsymbol{M[\langle a\rangle] = a \frown \langle\rangle} \\ \boldsymbol{M[X] = F[M[LP[X]], M[RP[X]]]} \end{matrix} \right)$$

where LP[X] stands for the left part of X, RP[X] stands for the right part of X and we know these

functions.

Theory must contain:

$$\text{Definition}\Big["\text{Left part of a tuple}",\ \text{any}[a, b, X],$$
$$\begin{Bmatrix} (LP[\langle\rangle] = \langle\rangle) \\ LP[\langle a\rangle] = \langle a\rangle \end{Bmatrix}$$
$$LP[a \frown (b \frown X)] = a \frown LP[X]$$

$$\text{Definition}\Big["\text{Right part of a tuple}",\ \text{any}[a, b, X],$$
$$\begin{Bmatrix} RP[\langle\rangle] = \langle\rangle \\ RP[\langle a\rangle] = \langle a\rangle \end{Bmatrix}$$
$$RP[a \frown (b \frown X)] = b \frown RP[X]$$

4.3.2. Case when we do not know the function F and we synthetise it

Reduced Problem:

Problem specification:

$I_{\text{FMSort}}[Y_1, Y_2]:$ IsSorted$[Y_1]$, IsSorted$[Y_2]$

$O_{\text{FMSort}}[Y_1, Y_2, Y]: \begin{cases} (Y_1 \asymp Y_2) \approx Y \\ \text{IsSorted}[Y] \end{cases}$

The problem become simpler because there are more assumptions about the input and for the output we have the same requirement.

We start to prove the proposition:

$$\text{Proposition}\left["\text{reduced problem MergeSort(2)}",\ \underset{Y_1,Y_2}{\forall}\left(\begin{matrix}\text{IsSorted}[Y_1]\\ \text{IsSorted}[Y_2]\end{matrix}\right)\Longrightarrow\underset{Y}{\exists}\left(\begin{matrix}(Y_1 \asymp Y_2)\approx Y\\ \text{IsSorted}[Y]\end{matrix}\right)\right]$$

using the induction principle:

$$\left(P[\langle\rangle,\langle\rangle]\bigwedge\underset{a,X}{\forall}(P[X,\langle\rangle]\Longrightarrow P[a\frown X,\langle\rangle])\bigwedge\right.$$

$$\underset{b,Y}{\forall}(P[\langle\rangle,Y]\Longrightarrow P[\langle\rangle,b\frown Y])\bigwedge$$

$$\left.\underset{a,b,X,Y}{\forall}(P[X,Y]\Longrightarrow P[a\frown X,b\frown Y])\right)$$

$$\Longrightarrow\underset{X,Y}{\forall}P[X,Y]$$

For proving the proposition "reduced problem MergeSort(2)" we assume:

IsSorted$[Y_1]$ (H3) and IsSorted$[Y_2]$ (H4) and we prove $\underset{Y}{\exists}\left(\begin{matrix}(Y_1 \asymp Y_2)\approx Y\\ \text{IsSorted}[Y]\end{matrix}\right)$ (G4)

Because we have two variables Y_1 and Y_2 we have 4 cases:

1) $Y_1 : \langle\rangle$ and $Y_2 : \langle\rangle$
2) $Y_1 : a \frown Y_1$ and $Y_2 : \langle\rangle$
3) $Y_1 : \langle\rangle$ and $Y_2 : b \frown Y_2$
4) $Y_1 : a \frown Y_1$ and $Y_2 : b \frown Y_2$

Case 1) $Y_1 : \langle\rangle$ and $Y_2 : \langle\rangle$

Prove: $\underset{Y}{\exists}\left(\begin{matrix}(\langle\rangle \asymp \langle\rangle)\approx Y\\ \text{IsSorted}[Y]\end{matrix}\right)$ (G4.1)

Find $Y^?$ such that $\begin{cases} (\langle\rangle \asymp \langle\rangle) \approx Y^? & \text{(G4 .11)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4 .12)} \end{cases}$

By definition of "concatenation" this is $\begin{cases} \langle\rangle \approx Y^? & \text{(G4 .13)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4 .12)} \end{cases}$

By matching $\langle\rangle \approx \langle\rangle$ (from the definition "$\approx$") with $\langle\rangle \approx Y^?$ we obtain witness $Y^? = \langle\rangle$.

Thus, we obtain $F[\langle\rangle,\langle\rangle]=\langle\rangle$. *(■)*

Check (G4.12): IsSorted[$\langle\rangle$]. This is true by the definition of "IsSorted".

Case 2) $Y_1 : a \frown Y_1$ *and* $Y_2 : \langle\rangle$

Assume: IsSorted[$a \frown Y_1$] (H4.2)

Prove: $\exists_Y \left(\begin{matrix} ((a \frown Y_1) \asymp \langle\rangle) \approx Y \\ \text{IsSorted}[Y] \end{matrix} \right)$ (G4.2)

Find $Y^?$ such that $\begin{cases} ((a \frown Y_1) \asymp \langle\rangle) \approx Y^? & \text{(G4 .21)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4 .22)} \end{cases}$

By definition of "concatenation" this is $\begin{cases} (a \frown Y_1) \approx Y^? & \text{(G4 .23)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4 .22)} \end{cases}$

By proposition "reflexivity in $\approx$" in (G4.23) we obtain witness $Y^? = a \frown Y_1$.

So, we have $F[a \frown Y_1, \langle\rangle]=a \frown Y_1$. *(■■)*

Check for (G4.22): IsSorted[$a \frown Y_1$]. This is true by (H4.2).

Theory must contain the proposition:

$\text{Proposition}\left["\text{reflexivity in } \approx", \ \forall_X (X \approx X)\right]$

Case 3) $Y_1 : \langle\rangle$ *and* $Y_2 : b \frown Y_2$

Assume : IsSorted[$b \frown Y_2$] (H4 .3)

Prove : $\exists_Y \begin{pmatrix} ((\langle\rangle \asymp (b \frown Y_2)) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G4 .3)

Find witness $Y^?$ such that $\begin{cases} ((\langle\rangle \asymp (b \frown Y_2)) \approx Y^? & \text{(G4 .31)} \\ \text{IsSorted}[Y^?] & \text{(G4 .32)} \end{cases}$

By definition of "concatenation" this is $\begin{cases} (b \frown Y_2) \approx Y^? & \text{(G4 .33)} \\ \text{IsSorted}[Y^?] & \text{(G4 .32)} \end{cases}$

By property " reflexivity in $\approx$ " in (G4 .33) we find witness $Y^? = b \frown Y_2$.

So, we have $F[b \frown Y_2, \langle\rangle] = b \frown Y_2$.(■■■)

Check for (G4 .32) : IsSorted[$b \frown Y_2$]. This is true by (H4 .3).

Case 4) $Y_1 : a \frown Y_1$ *and* $Y_2 : b \frown Y_2$

Assume: IsSorted[$a \frown Y_1$] (H4.41) and IsSorted[$a \frown Y_2$] (H4.42)

Prove: $\exists_Y \begin{pmatrix} ((a \frown Y_1) \asymp (b \frown Y_2)) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G5)

We consider that $(a \frown Y_1) \asymp Y_2$ and $Y_1 \asymp (b \frown Y_2)$ are smaller than $(a \frown Y_1) \asymp (b \frown Y_2)$ and for proving (G5) we use the induction principle:

$$\forall_{a,b,Y_1,Y_2} (P[(a \frown Y_1) \asymp Y_2] \bigwedge P[Y_1 \asymp (b \frown Y_2)]) \Longrightarrow \forall_{a,b,Y_1,Y_2} P[(a \frown Y_1) \asymp (b \frown Y_2)]$$

We reduce our problem into proving the proposition:

$$\text{Proposition}\Big["\text{reduced problem Merge 2.1}",\ \forall_{Y_1,Y_2} \forall_{a,b} \left(\left(\exists_{Z_1} \begin{pmatrix} (Y_1 \asymp (b \frown Y_2)) \approx Z_1 \\ \text{IsSorted}[Z_1] \end{pmatrix}\right) \bigwedge \exists_{Z_2} \begin{pmatrix} ((a \frown Y_1) \asymp Y_2) \approx Z_2 \\ \text{IsSorted}[Z_2] \end{pmatrix}\right) \Longrightarrow \exists_Z \begin{pmatrix} ((a \frown Y_1) \asymp (b \frown Y_2)) \approx Z \\ \text{IsSorted}[Z] \end{pmatrix}\Big)\Big]$$

Again, by reducing the problem into a simpler problem and by applying the same method we add asumptions to the input. So, the problem is simpler and the knowledge is extended.

For proving this we assume:

$\exists_{Z_1} \begin{cases} (Y_1 \asymp (b \frown Y_2)) \approx Z_1 \\ \text{IsSorted}[Z_1] \end{cases}$ (H5) and $\exists_{Z_2} \begin{cases} ((a \frown Y_1) \asymp Y_2) \approx Z_2 \\ \text{IsSorted}[Z_2] \end{cases}$ (H6) and

Prove: $\exists_Z \begin{cases} ((a \frown Y_1) \asymp (b \frown Y_2)) \approx Z \\ \text{IsSorted}[Z] \end{cases}$ (G5)

In (H5) and (H6) by Skolem we know:

$\begin{cases} (Y_1 \asymp (b \frown Y_2)) \approx Z_1 & \text{(H5 .1)} \\ \text{IsSorted}[Z_1] & \text{(H5 .2)} \end{cases}$ and $\begin{cases} ((a \frown Y_1) \asymp Y_2) \approx Z_2 & \text{(H6 .1)} \\ \text{IsSorted}[Z_2] & \text{(H6 .2)} \end{cases}$

Find witness $Z^?$ such that $\begin{cases} ((a \frown Y_1) \asymp (b \frown Y_2)) \approx Z^? & \text{(G5)} \\ \text{IsSorted}[Z^?] & \text{(G6)} \end{cases}$

Alternative 1:

Using the definition "concatenation" in (G5) we obtain $a \frown (Y_1 \asymp (b \frown Y_2)) \approx Z^?$ (G5.1)

By (H5.1) and by matching with the proposition "reflexivity in ≈" we obtain

$Z^? = a \frown (Y_1 \asymp (b \frown Y_2))$ (G5.2), and by (H5.1) we obtain the witness:

$Z^? = a \frown Z_1$ (∘)

Check for (G6): IsSorted$[a \frown Z_1]$

By the proposition "element in concatenation" and by (H5.1) we obtain that $b \lhd Z_1$ (H5.3).

By (H5.3), (H5.2) iff a≤b and if we consider that b is the smallest element from the sorted tuple Z_1 then we can use the proposition "Sorting II" and obtain that

IsSorted$[a \frown Z_1]$ is true, that was our goal.

So, we obtain: $a \leq b \Longrightarrow Z^? = a \frown Z_1$ and $F[a \frown Y_1, b \frown Y_2] = a \frown Z_1$ (■■■■)

Theory must contain the proposition:

$$\text{Proposition}\left["\text{element in concatenation}", \underset{a,A,B,C}{\forall} \begin{pmatrix} ((A \asymp (a \frown B)) \approx C) \Longrightarrow (a \lhd C) \\ (((a \frown A) \asymp B) \approx C) \Longrightarrow (a \lhd C) \end{pmatrix}\right]$$

$$\text{Proposition}\left["\text{Sorting II}", \underset{a,b,X}{\forall} ((\text{IsSorted}[b \frown X] \bigwedge (b \lhd (b \frown X)) \bigwedge (a \leq b)) \Longrightarrow \text{IsSorted}[a \frown (b \frown X)])\right]$$

Alternative 2:

We match the proposition "reflexivity in ≈" with (G5) and obtain the witness:

$Z^? = (a \frown Y_1) \asymp (b \frown Y_2)$.

Using the proposition "the same elements in concatenation" we obtain

$Z^? = b \frown ((a \frown Y_1) \asymp Y_2)$.

By (H6.1) we obtain the witness: $Z^? = b \frown Z_2$ (∘∘)

Check for (G6): IsSorted$[a \frown Z_2]$

By the proposition "element in concatenation" and by (H6.1) we obtain that $a \lhd Z_2$ (H6.3).

By (H6.3), (H6.2) iff b<a (using proposition "order") and if we consider that a is the smallest element from Z_2 then we can use the proposition "Sorting II" and obtain that

IsSorted$[b \frown Z_2]$ is true, that was our goal.

So, we obtain: $b < a \Longrightarrow Z^? = b \frown Z_2$ and $F[a \frown Y_1, b \frown Y_2] = b \frown Z_2$ (■■■■■)

Theory must contain the propositions:

Proposition["the same elements in concatenation", $\forall_{a,X,Y} (X \asymp (a \frown Y) \approx a \frown (X \asymp Y))$]

Proposition["order", $\forall_{a,b} \begin{pmatrix} (\neg (a \preceq b)) \Longrightarrow (b < a) \\ (b < a) \Longrightarrow (b \neq a) \end{pmatrix}$]

From ***(■), (■■), (■■■), (■■■■) and (■■■■■),*** by the transformation rules ($Z_1 \longrightarrow F[Y_1, b \frown Y_2], Z_2 \longrightarrow F[a \frown Y_1, Y_2]$) we obtain the algorithm:

$$\forall_{a,b,Y_1,Y_2} \left(\begin{array}{c} F[\langle\rangle, \langle\rangle] = \langle\rangle \\ F[a \frown Y_1, \langle\rangle] = a \frown Y_1 \\ F[\langle\rangle, b \frown Y_2] = b \frown Y_2 \\ F[a \frown Y_1, b \frown Y_2] = \left\{ \begin{array}{c} a \frown F[Y_1, b \frown Y_2], \ a \preceq b \\ b \frown F[a \frown Y_1, Y_2], \ \text{otherwise} \end{array} \right. \end{array} \right)$$

And by (*), (**), (***) and by the transformation rules:
$Y_1 \longrightarrow$M[LP[X]], $Y_2 \longrightarrow$M[RP[X]], $Y_1 \asymp Y_2 \longrightarrow$X
we obtain the algorithm for sorting:

$$\left\{ \begin{array}{c} \mathbf{M[\langle\rangle] = \langle\rangle} \\ \mathbf{M[}\boldsymbol{a} \frown \mathbf{\langle\rangle] = }\boldsymbol{a} \frown \mathbf{\langle\rangle} \\ \mathbf{M[}\boldsymbol{X}\mathbf{] = }\boldsymbol{F}\mathbf{[M[LP[}\boldsymbol{X}\mathbf{]], M[RP[}\boldsymbol{X}\mathbf{]]]} \end{array} \right.$$

The algorithm for F corresponds to "Merge" that combines the sorted parts of a tuple and the algorithm M is the algorithm for "Merge-Sort".

Note how by successive transformations of the problem we are able to reduce it to find witnesses which contain only elementary predicates and functions on tuples and on elements.

Chapter 5

Conclusions and Future work

We presented in this paper some experiments on synthesis of some sorting algorithms for tuples. We gave a short introduction of theory exploration and algorithm synthesis. In chapter 3 we presented in general the method that we use in order to synthesize the sorting algorithms (insertion-sort and merge-sort) and the experiments on the sorting algorithms in the case when we guess the witness.

In chapter 4 we presented in general the method that we use for synthesize the sorting algorithms (insertion-sort and merge-sort) and the experiments on the sorting algorithms in the case when we find the witness. These experiments from chapter 3 and from chapter 4 are paralleled with building the appropriate theory of tuples.

In our approach the problem was reduced into simpler and simpler problems and we applied the same method like in a "cascade". During the proof we introduce in the knowledge base the propositions that we need to continue the proof, in this way the process of proving is paralleled to the process of theory exploration.

The algorithm synthesis method presented here and the underlying proof methods are subject to implementation in the *Theorema* system. These case studies show that this method is effective and relatively efficient – the efficiency depends of course on the strength of the proof methods.

We plan to extend this work by investigating further algorithms in a similar way (like e. g. quick-sort).

The investigation of all these cases can constitute the basis of realizing a comprehensive theory of tuples, as well as a prover which is able to synthesize most of the algorithms, as well as to prove most of the conjectures. The work can then proceed in the direction of designing strategies for finding appropriate induction principles in the course of the synthesis process.

Bibliography

[1] P. Audebaud and L. Chiarabini. New Development in Extracting Tail Recursive Programs from Proofs. In *Pre-Proceedings of the 19th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'09)*, Coimbra, Portugal, September 9-11, 2009, 2009.

[2] D. Basin, Y. Deville, P. Flener, A. Hamfelt, J. F. Nilsson, and Mathematical Modelling. Synthesis of Programs in Computational Logic. In *Program Development in Computational Logic*, pages 30–65. Springer, 2004.

[3] B. Buchberger. Theory Exploration with Theorema. *Analele Universitatii Din Timisoara, Ser. Matematica-Informatica*, XXXVIII:9–32, 2000.

[4] B. Buchberger. Algorithm Invention and Verification by Lazy Thinking. *Analele Universitatii din Timisoara, Seria Matematica - Informatica*, XLI:41–70, 2003. special issue on Computer Science - Proceedings of SYNASC'03.

[5] B. Buchberger. Algorithm Supported Mathematical Theory Exploration: A Personal View and Strategy. In J. Campbell B. Buchberger, editor, *Proceedings of AISC 2004*, volume 3249 of Springer LNAI, pages pages 236–250, 2004.

[6] B. Buchberger and A. Craciun. Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. *Electr. Notes Theor. Comput. Sci.*, 93:24–59, 2004.

[7] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[8] A. Bundy, L. Dixon, J. Gow, and J. Fleuriot. Constructing Induction Rules for Deductive Synthesis Proofs. *Electron. Notes Theor. Comput. Sci.*, 153(1):3–21, 2006.

[9] L. Chiarabini. Extraction of Efficient Programs from Proofs: The Case of Structural Induction over Natural Numbers. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *Local Proceedings of the Fourth Conference on Computability in Europe: Logic and Theory of Algorithms (CiE'08)*, pages 64–76, Athens, Greece, June 15-20, 2008, 2008.

[10] A. Craciun and B. Buchberger. Algorithm Synthesis Case Studies: Sorting of Tuples by Lazy Thinking. Technical Report 04-16, RISC–Linz, Austria, October 2004.

[11] A. Craciun and M. Hodorog. Decompositions of Natural Numbers: From A Case Study in Mathematical Theory Exploration. In D. Petcu, D. Zaharie, V. Negru, and T. Jebelean, editors, *SYNASC07*, 2007.

[12] J. Darlington. A Synthesis of Several Sorting Algorithms. *Acta Inf.*, 11:1–30, 1978.

[13] I. Dramnesc, T. Jebelean, and A. Craciun. Case Studies in Systematic Exploration of Tuple Theory. Technical Report 10-09, RISC Report Series, University of Linz, Austria, May 2010.

[14] M. Hodorog and A. Craciun. A Case Study in Systematic Theory Exploration: Natural Numbers. Technical Report 07-18, RISC–Linz, Austria, October 2007.

[15] Tudor Jebelean. Reverse of a list: A Simple Case Study in Lisp Programming, March 2009.

[16] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching*. Addison-Wesley Longman Publishing Co.,Inc., Redwood City, CA, USA, 1998.

[17] I. Kraan, D. Basin, and A. Bundy. Middle-Out Reasoning for Logic Program Synthesis. In *Proc. 10th Intern. Conference on Logic Programing (ICLP '93)* (Budapest, Hungary), pages 441–455, Cambridge, MA, 1993. MIT Press. Also available as Technical Report MPI-I-93-214.

[18] K. K. Lau. A Note on Synthesis and Classification of Sorting Algorithms. *Acta Informatica*, 27:73–80, 1989.

[19] K. K. Lau and G. Wiggins. A Tutorial on Synthesis of Logic Programs from Specifications. In P. Van Hentenryck, editor, *Proc. 11th Int. Conf. on Logic Programming*, pages 11–14. MIT Press, 1994.

[20] Z. Manna and R. Waldinger. Synthesis: Dreams ? programs. *IEEE Transactions on Software Engineering*, 5:294–328, 1979.

[21] Susan M. Merritt. An Inverted Taxonomy of Sorting Algorithms. *Commun. ACM*, 28(1):96–99, 1985.

[22] D. R. Smith. A Problem Reduction Approach to Program Synthesis. In *IJCAI*, pages 32–36, 1983.

[23] D. R. Smith. Top-down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, 27(1):43–96, 1985. (Reprinted in "Readings in Artificial Intelligence and Software Engineering", Eds. C. Rich and R. Waters, Morgan Kaufmann Pub. Co., Los Altos, CA, 1986, pp.35-61).

Appendix

By convention in this paper the definitions are written in the *Theorema* system.

The *Knowledge Base* contain:

$$\textit{Definition}\Big[\,\textit{"concatenation"}, \textit{any}[a, X, Y],$$
$$\begin{array}{c} \langle\rangle \asymp X = X \\ (a \smile X) \asymp Y = a \smile (X \asymp Y) \end{array}\,\Big]$$

$$\textit{Definition}\Big[\,\textit{"}\lhd\textit{"}, \textit{any}[a, b, X],$$
$$\begin{array}{c} a \not\lhd \langle\rangle \\ a \lhd a \smile X \\ (a \neq b) \Longrightarrow ((a \lhd b \smile X) \Longleftrightarrow (a \lhd X)) \end{array}\,\Big]$$

$$\textit{Definition}\Big[\,\textit{"dfo"}, \textit{any}[a, b, X],$$
$$\begin{array}{c} \textit{dfo}[a, \langle\rangle] = \langle\rangle \\ \textit{dfo}[a, a \smile X] = X \\ (a \neq b) \Longrightarrow (\textit{dfo}[a, b \smile X] = b \smile \textit{dfo}[a, X]) \end{array}\,\Big]$$

$$\textit{Definition}\Big[\,\textit{"}\approx\textit{"}, \textit{any}[a, X, Y],$$
$$\begin{array}{c} \langle\rangle \approx \langle\rangle \\ ((a \smile X) \approx Y) \Longleftrightarrow ((a \lhd Y) \wedge X \approx \textit{dfo}[a, Y]) \end{array}\,\Big]$$

$$\textit{Definition}\Big[\,\textit{"IsSorted"}, \textit{any}[a, b, X],$$
$$\begin{array}{c} \textit{IsSorted}[\langle\rangle] \\ \textit{IsSorted}[a \smile \langle\rangle] \\ \textit{IsSorted}[a \smile (b \smile X)] \Longleftrightarrow ((a \leq b) \wedge \textit{IsSorted}[b \smile X]) \end{array}\,\Big]$$