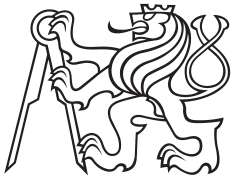
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra kybernetiky

Závislé typy a specifikace v teorii typů

Marika Slouková

Vedoucí: Ing. Matěj Dostál, Ph.D.
Studijní program: Otevřená informatika
Specializace: Základy umělé inteligence a počítačových věd
Květen 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Slouková Marika** Personal ID number: **494807**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Dependent Types and Specification in Type Theory

Bachelor’s thesis title in Czech:

Závislé typy a specifikace v teorii typů

Guidelines:

The question of how to develop provably correct software arises in theoretical as well as practical computer science. One of the more successful methods is to use a programming language whose type system is strong enough to allow the following:
1. Specification of program behaviour.
2. Proving that a given program meets the desired specification.
The student will familiarize themselves with the Coq proof assistant [4] which is used for the above tasks in practice [2]. She will learn basic methods of program specification: weak specification and strong specification [1]. She will choose (at least) two algorithms/data structures (e.g. insertion sort, selection sort, merge sort, binary search tree), implement them in Coq and prove their correctness (or in the case of a data structure, prove the correctness of the operations pertaining to the structure).
The student is not expected to cover all the theoretical aspects of dependent type theory in the thesis but rather to explain the basic motivation of using dependent types in an accessible way.

Bibliography / sources:

[1] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development: Coq’Art: The Calculus of Inductive Constructions, Springer Science & Business Media, 2004
[2] A. Chlipala, Certified Programming with Dependent Types, available online: http://adam.chlipala.net/cpdt/
[3] B. J. Pierce et al., Logical Foundations, available online: https://softwarefoundations.cis.upenn.edu/lf-current/index.html
[4] Coq reference manual, available online: https://coq.inria.fr/doc/V8.17.1/refman/

Name and workplace of bachelor’s thesis supervisor:

Ing. Matěj Dostál, Ph.D. Department of Mathematics FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **26.01.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Matěj Dostál, Ph.D.	prof. Dr. Ing. Jan Kybic	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Poděkování

Děkuji mému vedoucímu Ing. Matěji Dostálovi, Ph.D. za vedení mé bakalářské práce a podporu v průběhu jejího psaní. Také děkuji mé rodině a blízkým za podporu během mých studií.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 24. května 2024

Abstrakt

Vývoj softwaru, který je nejen funkční, ale i formálně prokazatelně korektní, je klíčovou výzvou v informatice. V této bakalářská práce se zaměřujeme na formální dokazování korektnosti algoritmů pomocí důkazového asistenta Coq a jazyka Gallina. Po úvodním zkoumání teoretických základů a významu silných typových systémů implementujeme dva třídicí algoritmy, Insertion sort a Merge sort, v jazyce Coq.

Provádíme formální důkaz jejich korektnosti a porovnáváme tento proces s klasickými metodami dokazování. Použití Coqu poskytuje vysokou míru jistoty ohledně správnosti implementace, což je zásadní pro kritické aplikace. Oproti tomu klasické metody dokazování poskytují hlubší vhled do algoritmických principů.

Práce přispívá k lepšímu pochopení formálních metod v softwarovém inženýrství a ilustruje jejich praktické použití.

Klíčová slova: Coq, Slabá specifikace, Silná specifikace, Insertion sort, Merge sort, Korektnost algoritmů

Vedoucí: Ing. Matěj Dostál, Ph.D.

Abstract

The development of software that is not only functional but also formally provably correct is a key challenge in computer science. This bachelor's thesis focuses on the formal verification of algorithm correctness using the Coq proof assistant and the Gallina language. After an initial examination of theoretical foundations and the significance of strong type systems, two sorting algorithms, Insertion sort and Merge sort, are implemented in Coq.

Formal proofs of their correctness are conducted and compared with classical methods of verification. The use of Coq provides a high level of certainty regarding the correctness of implementation, which is crucial for critical applications. On the other hand, classical methods of verification offer a deeper insight into algorithmic principles.

This work contributes to a better understanding of formal methods in software engineering and demonstrates their practical application.

Keywords: Coq, Weak specification, Strong specification, Insertion sort, Merge sort, Algorithm correctness

Title translation: Dependent Types and Specification in Type Theory

Obsah

Kapitola 1

Úvod

Vývoj softwaru, který je nejen funkční, ale i formálně prokazatelně korektní, představuje klíčovou výzvu v oblasti informatiky. Otázka, jak takový software vyvinout, je relevantní jak v teoretické, tak v praktické informatice. V praxi se setkáváme s řadou metod, které nám umožňují ověřit korektnost softwaru, avšak jednou z nejúspěšnějších je použití programovacího jazyka s dostatečně silným typovým systémem. Tento typový systém nám umožňuje nejen specifikovat chování programu, ale také formálně dokázat, že daný program splňuje požadovanou specifikaci.

Tato práce se zaměřuje na formální dokazování korektnosti pomocí důkazového asistenta Coq s jazykem Gallina, který splňuje všechny zmíněné vlastnosti. Naučíme se pracovat s tímto důkazovým asistentem, jeho jazykem a taktikami. Seznámíme se se základními metodami specifikace programu, konkrétně slabou a silnou specifikací, a následně aplikujeme jednu z metod na konkrétní algoritmy.

Konkrétně v této práci implementujeme dva algoritmy, a to Insertion sort a Merge sort. Provedeme formální důkaz jejich korektnosti v Coqu a formální důkaz jejich korektnosti klasicky a ukážeme rozdílný přístup v těchto důkazech.

Kapitola 2
Úvod do Coq

Coq je interaktivní důkazový asistent, který je založen na teorii závislých typů a patří mezi nejmocnější nástroje pro formální verifikaci. Jeho prostředí umožňuje vytvářet a ověřovat formální matematické důkazy a programy, přičemž práce probíhá ve funkcionálním jazyce Gallina. Coq nabízí širokou škálu funkcí pro práci s důkazy a podporuje interaktivní dokazování, automatizované techniky dokazování a extrakci ověřeného kódu do různých programovacích jazyků, jako jsou OCaml, Haskell nebo Scheme. Používá se při formálním ověřování softwaru, formulaci matematických tvrzení a výuce formální logiky a teorie důkazů.

Kvůli těmto vlastnostem jsme si k dokázání korektnosti řadících algoritmů v této práci vybrali právě Coq.

V této kapitole si ukážeme práci v Coqu. Předvedeme si definice typů a funkcí, zavádění lemmat, obecnou strukturu důkazů a základní taktiky jejich dokazování [CP24, App23].

Ve standardní knihovně Coqu nalezneme již zadefinované typy jako jsou booleany, seznamy nebo přirozená čísla. Pojďme si ale ukázat, jak taková definice vypadá. Induktivně si zadefinujeme boolean.

```
Inductive bool : Type :=
  | true
  | false.
```

Typ se jmenuje `bool` a jeho členy jsou `false` a `true`. Nyní si zadefinujeme nějakou funkci, která pracuje s definicí `bool`.

```
Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.
```

Funkce `negb` je klasická negace, která pro vstup `true` vratí `false` a naopak. Typ argumentu funkce a typ návratové hodnoty jsou explicitně uvedeny. Coq je ale dost chytrý na to, aby tyto typy uhodl. V těle funkce vidíme slova

`match` a `with`, která indikují pattern matching. Koukáme pak na proměnnou `b` a zjišťujeme, kterým konstruktorem svého typu vznikla. Podle příslušné větve vrátí funkce příkaz za šipkou.

Nyní novou funkci použijeme v příkladech. Můžeme pro to použít klíčové slovo `Compute` a výraz spočítá.

```
Compute (negb true).
```

Po evaluaci dostaneme výpis `false`, jak jsme očekávali.

```
= false
: bool
```

Příklady můžeme psát i s klíčovým slovem `Example`. V takovém případě vždy píšeme, čemu očekáváme, že se daný výraz rovná, a poté takové tvrzení musíme i dokázat.

```
Example test_negb_false:
  (negb (negb false)) = false.
Proof.
  simpl. reflexivity.
Qed.
```

Každý příkaz v Coqu končí tečkou, ať už je to definice funkce začínající slovem `Definition`, nebo induktivní definice typu `Inductive`. Všimněme si, že i `Example` a `Compute` končí tečkou jako každý jiný příkaz.

Každý důkaz v Coqu začíná klíčovým slovem `Proof` a končí slovem `Qed`. Obě tato slova jsou příkazy a musí tedy za nimi být tečka. Slovo `Proof` značí začátek důkazu. V levé části prostředí se nám zobrazí cíl, který musíme dokázat.

```
1 goal
                                                (1/1)
negb (negb false) = false
```

Použijeme taktiku `simpl`, kterou Coq zjednoduší všechny výrazy v cíli. Dokazování v Coqu nefunguje stejně jako na papíře, kdy máme jeden cíl, který se nemění a pomocí rovnic rovnic, implikací a úvah se dobereme k onomu cíli. Zde se cíl v průběhu důkazu mění podle toho, jaké taktiky jsme použili. Po použití taktiky `simpl` se výraz `negb (negb false)` zjednoduší na výraz `false`.

```
1 goal
                                                (1/1)
false = false
```

Poté použijeme taktiku `reflexivity`, čímž dokážeme cíl. Jelikož tohle byl náš jediný cíl, dokázali jsme vše, co jsme chtěli dokázat. Taktika `reflexivity` je důkaz reflexivity ekvivalence rovná se, tedy klasická reflexivita, kterou známe. Používáme ji k důkazu rovnosti dvou výrazů.

```
No more goals.
```

Důkaz zakončíme příkazem `Qed.` a levá část, kde jsme sledovali průběh důkazu, se vyprázdní.

Zadefinujme si přirozená čísla. Nule odpovídá `O`, jedničce `S O`, dvojce `S (S O)` a tak dále.

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

Zadefinujme si funkci plus. Jelikož ji budeme definovat rekurzivně, použijeme klíčové slovo `Fixpoint`, které na tuto skutečnost upozorňuje.

```
Fixpoint plus (n : nat) (m : nat) : nat :
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

Můžeme se podívat, jakého typu je funkce `plus` pomocí příkazu `Check plus.`.

```
Check plus.
```

```
plus
     : nat -> nat -> nat
```

Funkce `plus` je funkce o dvou agrumentech typu přirozené číslo, v Coqu `nat`, která vrací jedno přirozené číslo. Funkci `plus` nevnímáme jako funkci, ale jako binární operaci. Můžeme zavést notaci, která se bude podobat zápisu, na který jsme s čísly zvyklí. Poté nebudeme muset `plus` používat s prefixním člověku nelibým zápisem, ale s infixním nám přirozeným.

```
Notation "x + y" := (plus x y).
```

```
Compute (plus 4 5). Compute (4 + 5).
```

Kromě příkladů můžeme v Coqu definovat i lemmata a teorémy. Zadefinujme si teorém o sčítání dvou čísel a ukažme si další důležité taktiky.

```
Theorem plus_id_example : forall n m:nat,
  n = m ->
  n + n = m + m.
Proof.
  intros n m.
  intros H.
  rewrite -> H.
  reflexivity. Qed.
```

Všimněme si klíčového slova `forall` který odpovídá logické značce pro všechna. Teorém tedy říká, že pro všechna přirozená čísla `n` a `m` platí, že pokud se `n` rovná `m`, pak `n + n` se rovná `m + m`. Tohle tvrzení dokážeme následovně. Označíme počátek důkazu.

```
1 goal
______________________________________(1/1)
forall n m : nat, n = m -> n + n = m + m
```

Pomocí taktiky `intros n m` zvolíme libovolné pevné `n` a `m` a cíl se nám zjednoduší. Tato taktika převede všechny předpoklady v cíli do místa pro předpoklady, a to nad čarou.

```
1 goal
n, m : nat
______________________________________(1/1)
n = m -> n + n = m + m
```

Takže když ji použijeme podruhé, objeví je nám předpoklad `H` mezi ostatními předpoklady a zmizí z cíle.

```
1 goal
n, m : nat
H : n = m
______________________________________(1/1)
n + n = m + m
```

Nyní použijeme taktiku `rewrite -> H`. Tato taktika nahradí veškeré výskyty levé strany v cíli za pravou. Tedy nahradí každé `n` za `m`. Šipka doleva, která strana se má nahradit za kterou, jestli levá za pravou, či pravá za levou. Písmeno `H` značí, podle které rovnosti se bude nahrazovat. Mohli bychom totiž použít rovnost definovanou v předem dokázaném teorému nebo lemmatu.

```
1 goal
n, m : nat
H : n = m
______________________________________(1/1)
m + m = m + m
```

Cíl se zase změnil a dále postupujeme s taktikami, které již známea důkaz dokončíme.

Poďme si zadefinovat další teorém o dvojité negaci a ukázat si novou taktiku.

Po taktice `intros` použijeme taktiku `destruct`. Tato taktika rozdělí cíl na tolik cílů podle toho, jak mohlo vzniknout `b`. Proměnná `b` mohla podle definice boolu vzniknout pomocí jednoho z dvou konstruktorů, tedy dvěma způsoby. Buď je `b true`, nebo `false`. Slovo `eqn:` zapíše aktuální rovnost proměnné `b` do předpokladů a pojmenuje ji `E`.

První cíl dokážeme pomocí známé `reflexivity`. Tato taktika je mnohem mocnější. Umí porovnávat dva výrazy, i když zrovna ještě nevypadají stejně,

```
Theorem negb_involutive : forall b : bool,
  negb (negb b) = b.
Proof.
  intros b. destruct b eqn:E.
  - reflexivity.
  - reflexivity. Qed.

2 goals
b : bool
E : b = true
______________________________________(1/2)
negb (negb true) = true
______________________________________(2/2)
negb (negb false) = false
```

pokud se oba výrazy dají zjednodušit tak, že vypadají stejně. Dokázali jsme první cíl.

```
1 goal
b : bool
E : b = false
______________________________________(1/1)
negb (negb false) = false
```

Poté se přesuneme do cíle druhého. Všimněme si, že předpoklad `E` se změnil podle větve, ve které se zrovna nacházíme. Tento cíl rovněž dokážeme `reflexivitou`. Oba cíle jsme dokázali, žádné další nemáme, důkaz je hotov.

Nyní se podíváme na další taktiku a to `induction`. Tato taktika funguje podobně jako předchozí taktika. Rozdělí cíl na tolik cílů, jako je konstruktorů v definici typu proměnné, podle které se indukuje.

```
Theorem add_0_r : forall n:nat, n + 0 = n.
Proof.
  intros n. induction n as [| n' IHn'].
  - reflexivity.
  - simpl. rewrite -> IHn'. reflexivity. Qed.
```

Dále funguje tak, jak byste si představovali, že funguje klasická indukce. Proměnnou `n` v cílích nahradí výrazem podle příslušného konstruktoru. Přičemž v poslední větvi přidá do předpokladů induktivní předpoklad.

```
2 goals
______________________________________(1/2)
0 + 0 = 0
______________________________________(2/2)
S n' + 0 = S n'
```

Tato taktika je silnější než `destruct` právě o indukční předpoklad, bez kterého bychom cíl nedokázali. Cíl následně zjednodušíme a použijeme indukční předpoklad taktikou `rewrite`.

```
1 goal
n' : nat
IHn' : n' + 0 = n'
______________________________________(1/1)
S n' + 0 = S n'
```

Další taktiku, kterou si zde zmíníme, je taktika `apply H`. Na první pohled by se mohla plést s taktikou `rewrite`, která nahrazuje výskyty výrazu `a` za výraz `b` z předpokladu nebo již dokázaného lemmatu. Následně pak ale musíme použít taktiku `reflexivity`. Taktika `apply`, jak se sama překládá, použije předpoklad nebo již dokázané lemma `H` na cíl a tím jej dokáže.

```
Theorem silly1 : forall (n m : nat),
  n = m -> n = m.
Proof.
  intros n m eq.
  apply eq. Qed.
```

Podívejme se na použití taktiky `apply` zde. Po zavedení `n`, `m` a předpokladu `eq` se v cíli objeví výraz stejný jako je `eq`. Tím, že máme v ruce předpoklad `eq`, tak máme v ruce jeho důkaz. A ten teď můžeme použít k dokázání cíle tím že napíšeme příkaz `apply eq`.

```
1 goal
n, m : nat
eq : n = m
______________________________________(1/1)
n = m
```

Představme si, že známe nějaké již dokázané lemma a chtěli bychom ho použít k důkazu cíle. Problém je v tom, že v cíli máme rovnost naopak, než v lemmatu. Tento problém snadno vyřešíme taktikou `symmetry`, která rovnost výrazů `x = y` převede na výraz `y = x`. Tato taktika se chová stejně, jak byste předpokládali, a to jako symetrie ekvivalence rovnosti.

Další důležitou taktikou je taktika `unfold fce`. Pokud nám při zjednodušování výrazu nepomůže taktika `simpl`, může nám pomoct taktika `unfold`. Tato taktika, jak sama napovídá, rozbalí zmíněnou funkci `fce` a my tak dostaneme možnost do ní nahlédnou a použít další taktiky. Zvláště se hodí v případech, kdy se v rozbalené funkce objeví nerovnost, na kterou následně budeme chtít použít `destruct`.

Výrazy, které se objevují v lemmatech, nemusí být jen složením funkcí a implikací. Jejich součástí mohou i být konjunkce a disjunkce. Pokud je tedy cíl konjunkcí dvou výrazů, použijeme taktiku `split`, která cíl rozdělí na dva cíle. Každou větev konjunkce pak budeme dokazovat zvlášť.

Ani disjunkce nás nepřekvapí, že bude fungovat tak, jak si představujeme. Pokud v cíli uvidíme disjunkci dvou výrazů, můžeme si vybrat, který z nich dokážeme. Buď pravý, a to použijeme taktiku `right`, a nebo levý, a

to analogicky použijeme taktiku `left`. Cíl se nám následně zjednoduší na vybraný výraz a můžeme pokračovat v dokazování.

Další důležitou a jistě vítanou taktikou je taktika `lia`. Tato taktika je velmi chytrá a udělá spoustu práce, kterou bychom dokazovali velmi zdlouhavě a krkolomně, udělá za nás. Slovo `lia` je zkratkou pro lineární aritmetiku. Taktika pracuje s celými a přirozenými čily, s rovnostmi a nerovnostmi, sčítáním a odčítám, s negací, s násobením malým číslem a občas i s konjunkcí, disjunkcí a ekvivalencí.

Takže pokud se nám předpokladech objeví spousta nerovností, které spolu souvisejí, a v cíli také nejaká nerovnost, použijeme taktiku `lia`.

Taktiky nemusíme používat jen na cíl. Často je potřeba zjednodušit předpoklad, abychom jej převedli na výraz, který se nám hodí. K tomu použijeme klíčové slovo `in`, například takto `apply H in H2`. Tímto jsme řekli, že aplikujeme předpoklad `H` na předpoklad `H2`.

Jelikož se v této práci budeme věnovat řezním seznamů, ukážeme si jak jsou seznamy definované základní knihovně Coqu.

```
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).
```

Seznam je definovaný induktivně tak, že senzam je buď prázdný seznam, nebo složení čísla a seznamu. Klíčové slovo `cons` představuje ono složení. Abychom mohli značit seznamy nám známým způsobem a nemuseli používat zápis z definice `cons (cons 4 (cons 6 (cons 2 nil)))`, zavedeme si dvě notace.

```
Notation "x :: l" := (cons x l)
                     (at level 60, right associativity).
```

Slova `at level` a `right associativity` značí Coqu, jak se k nim má chovat překladač. Tyto anotace bychom mohli přirovnat k informacím, které známe o přednostech v matematice. Prvek `x` bychom mohli označit za head a seznam `l` za tail.

```
Notation "[ ]" := nil.
```

Tato notace značí prázdný seznam tak, jak jsme zvyklí.

Na závěr si zadefinujeme funkci `app` a její notaci. Tato rekurzivní funkce nám umožňuje skládat dva seznamy za sebe, nebo spojovat dva seznamy v jeden.

Skládat za sebe můžeme i více seznamů, například `l1 ++ l2 ++ l3`. Ve skutečnosti se vždy spolu skládají pouze dva seznamy díky asociativitě. Výsledný seznam `l` složený z `l2` a `l3` se poté složí se seznamem `l1`.

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil ⇒ l2
  | h :: t ⇒ h :: (app t l2)
  end.
```

```
Notation "x ++ y" := (app x y)
                     (right associativity, at level 60).
```

Kapitola 3

Řadící algoritmy Insertion sort a Merge sort a jejich korektnost

3.1 Základní definice

Než se do samotného důkazu pustíme, zadefinujeme si pár potřebných věcí, se kterými budeme v onom důkazu pracovat. Na první pohled se může zdát, že jsou tyto věci intuitivní, ale i tak je potřeba je zadefinovat.

Začneme definicí seznamu. Mějme množninu A. *Seznam* prvků z A je konečná posloupnost prvků z A, kterou značíme $[a_1, a_2, a_3, ..., a_n]$, kde $n \in \mathbb{N}$. Tento zápis budeme chápat tak, že pro $n = 0$ budeme seznam značit jako [] a budeme mu říkat prázdný seznam. Pro $n = 1$ budeme seznam značit jako $[a_1]$ a budeme mu říkat jednoprvkový seznam. Délka seznamu je n, budeme značit *length* $l = n$ pro nějaký seznam l.

Multimnožina je zobrazení M z množiny A do množiny přirozených čísel $\mathbb{N}$.

Uveďme si příklad: Mějme množinu $A = \{a, b, c\}$ a M je multimnožina taková, že $M : A \to \mathbb{N}$, a platí $a \mapsto 3$, $b \mapsto 0$, $c \mapsto 1$. Takovou multimnožinu budeme značit $M = \{a : 3,\ b : 0,\ c : 1\}$.

Mějme multimnožinu $E : A \to \mathbb{N}$, kde $E = \{a : 0, b : 0, c : 0\}$. Zavedeme notaci, že všechny prvky a, pro něž platí $a \mapsto 0$, do zápisu multimnožiny psát nebudeme. Budeme tedy psát $E = \{\}$ a $M = \{a : 3,\ c : 1\}$. Multimnožinu E a každou multimnožinu $F = \{\}$ budeme nazývat prázdnou multimnožinou.

Zavedeme druhou notaci, a to takovou, že všechny prvky a, pro něž platí $a \mapsto 1$, v zápisu multimnožiny budeme jen jako a místo $a : 1$. Budeme tedy psát $M = \{a : 3,\ c\}$.

Dvě multimnožiny M a N, kde $N = \{a : 0,\ b : 1,\ c : 2\}$, budeme sčítat tak, že $M + N = \{a : 3 + 0,\ b : 0 + 1,\ c : 1 + 2\}$.

My si zadefinujeme elementy seznamu l, značeno *elems* l, jako multimnožinu, kde M nás bude informovat o tom, kolikrát se každý prvek z množiny prvků A vyskytuje v seznamu l.

Pro sjednocení elementů dvou seznamů l a k platí vztah *elems* $l \cup$ *elems* k $=$ *elems* $l +$ *elems* k.

Jelikož se tu bavíme o řadících algoritmech, měli bychom si také zadefinovat, co to znamená, když je seznam seřazený, *sorted*. Řekněme, že $A = \mathbb{N}$. Seznam $[a_1, ..., a_n]$ je seřazen, pokud je prázdný, nebo jednoprvkový. Dále pokud platí $a_i \leq a_{i+1}$ pro každé přirozené číslo $i < n$. Nebo $a_i \leq a_j$ pro každá přirozená čísla i a j, pro která platí $i < j \leq n$.

Tyto dvě definice seřazenosti jsou navzájem ekvivaletní, proto v důkazech budeme používat vždy tu, která se nám v dané chvíli bude více hodit. Že jsou tyto dvě definice navzájem ekvivaletní se dá jednoduše dokázat pomocí tranzitivity.

Z obou definic seřazenosti vidíme, že prázdný a jednoprvkový seznam je vždy seřazen.

Pro práci s páry si zadefinujeme dvě funkce, a to funkce *left* a *right*. Dělají přesně to, co byste od nich čekali, a tedy, že osvobozují levý a pravý prvek

páru. Mějme tedy pár r s prvky p a q. Jestliže pár $r = (p, q)$, potom *left* r = *left* $(p, q) = p$ a *right* r = *right* $(p, q) = q$.

3.2 INSERTION SORT - důkaz korektnosti algoritmu

Insertion sort je sortovací (řadící) algoritmus vkládáním, který lze jednoduše implementovat a také ověřit. Avšak svou jednoduchost si kompenzuje kvadratickou časovou složitostí $O(n^2)$.

Algoritmus funguje tak, že do seřazené posloupnosti prvků vloží nový prvek na správné místo tak, aby nová posloupnost byla také sežazená. Začíná s prázdnou posloupností, do které vloží první prvek. Poté do nové posloupnosti o prvním prvku vloží prvek druhý. A takto postupně vloží do seřazené poslupnosti všechny prvky z původní neseřazené posloupnosti.

V této sekci práce dokážeme korektnost tohoto algoritmu matematicky.

V našem případě použijeme rekurzivní implementaci v jazyce Gallina, jelikož se podobá implementaci v poslední kapitole, kde v této podobě budeme s algoritmem pracovat [Uni08].

```
Fixpoint insert (i : nat) (l : list nat) :=
  match l with
  | [] => [i]
  | h :: t => if i <=? h then i :: h :: t else h :: insert i t
  end.

Fixpoint sort (l : list nat) : list nat :=
  match l with
  | [] => []
  | h :: t => insert h (sort t)
  end.
```

Insertion sort algoritmus používá pomocnou funkci *insert*, která vloží prvek do seřazeného seznamu tak, že výsledkem bude zase sežazený seznam.

Funkce *sort* prochází seznam prvek po prvku a seznam řadí tím, že vkládá prvek do seřazeného seznamu pomocí funkce *insert*.

Dokážeme, že se algoritmus chová korekně pro libovolný vstup, tedy že výsledný seznam bude seřazený a bude obsahovat všechny prvky jako seznam do algoritmu vstupující.

Důkaz rozdělíme na dvě části, a to důkaz korektnosti funkce *insert* a důkaz korektnosti funkce *sort*. Obě funkce jsou rekurzivní, a tak se nabízí důkaz matematickou indukcí.

3.2.1 Důkaz korektnosti funkce *insert*

Chceme dokázat, že pro libovolné přirozené číslo i a pro libovolný seznam přirozených čísel l, který je seřazený a má délku n, platí, že výsledný seznam *insert* i l je také seřazený a obsahuje všechny prvky seznamu l a prvek e.

Chceme tedy dokázat tvrzení $P(n)$ pro každé $n \geq 0$, kde

$$P(n) = \forall \text{seznam } l, \text{prvek } e :$$
$$\text{if sorted}(l) \wedge \text{length}(l) = n$$
$$\text{then sorted}(\text{insert}(e, l)) \wedge \text{elems}(\text{insert}(e, l)) = \text{elems}(l) \cup \{e\}$$

Základní bod indukce: $n = 0$
Chceme dokázat $P(0)$, tedy

$$P(0) = \forall \text{seznam } l, \text{prvek } e :$$
$$\text{if sorted}(l) \wedge \text{length}(l) = 0$$
$$\text{then sorted}(\text{insert}(e, l)) \wedge \text{elems}(\text{insert}(e, l)) = \text{elems}(l) \cup \{e\}$$

Takový seznam existuje právě jediný, a to prázdný seznam *[]*.
Nechť tedy l je prádný seznam [] a i je libovolné přirozené číslo. Víme, že prázdný seznam má délku 0 a že je seřazený.

Po evaluaci (vyhodnocení) *insert i []* dostaneme *[i]*.
Každý jednoprvkový seznam je z definice *sorted* seřazený, takže i seznam *[i]* je seřazený, tedy *insert i l* je seřazený.
Dále platí

$$\text{elems}(\text{insert } i \ []) = \text{elems}([i]) = \{i\} = \{i\} \cup \emptyset = \{i\} \cup \text{elems}([]),$$

neboli *insert i []* obsahuje prvky seznamu [] a prvek i.
Tímto jsme dokázali základní krok indukce.

Induktivní krok: Nyní chceme dokázat, že pokud platí *P(n)*, pak platí i *P(n+1)*.
Předpokládejme, že *P(n)* platí pro libovolné pevné $n \geq 0$. Tedy, že platí

$$P(n) = \forall \text{seznam } l, \text{prvek } e :$$
$$\text{if sorted}(l) \wedge \text{length}(l) = n$$
$$\text{then sorted}(\text{insert}(e, l)) \wedge \text{elems}(\text{insert}(e, l)) = \text{elems}(l) \cup \{e\}$$

pro libovolné pevné $n \geq 0$.
Chceme dokázat, že platí *P(n+1)*, tedy

$$P(n+1) = \forall \text{seznam } l, \text{prvek } e :$$
$$\text{if sorted}(l) \wedge \text{length}(l) = n+1$$
$$\text{then sorted}(\text{insert}(e, l)) \wedge \text{elems}(\text{insert}(e, l)) = \text{elems}(l) \cup \{e\}$$

Nechť i je libovilné přirozené číslo a l je libovolný seznam přirozených čísel, pro který platí *sorted* l a *length* $l = n+1$. Seznam l můžeme napsat jako $h::t$, kde *sorted* t a *length* $t = n$, a můžeme říct, že *elems* l = *elems* $t \cup \{h\}$.

Po evaluaci *insert* i l dostaneme dva možné výstupy podle toho, jakou hodnotu vrátí $i \leq h$.

Pojďme si tyto dva případy rozebrat. Označme *insert* i l jako s.

a) $i \leq h$

insert i l se evaluuje na $i::h::t$.

Pokud $t = []$, pak $l = [h]$ a *insert* i l se evaluuje na $i :: h$. Jelikož $i \leq h$, pak z první definice sorted $s_j \leq s_{j+1}$ pro každé přirozené číslo $j < n$, $n = 2$, tedy $j = 1$, $s_j = j$ a $s_{j+1} = h$ platí, že s je seřazený, tedy že *insert* i l je seřazený.

Nechť t není prázdný seznam. Protože l je seřazený, pak platí $s_j \leq s_{j+1}$ pro každé přirozené číslo j, kde $1 < j < n + 2$. Protože $i \leq h$, pak i $s_j \leq s_{j+1}$ pro $j = 1$. Seznam s je seřazený, protože platí $s_j \leq s_{j+1}$ pro každé $j < n + 2$, tedy *insert* i l je seřazený.

Dále můžeme říct, že *insert* i l obsahuje prvky seznamu l a prvek i, protože *elems (insert i l)* = *elems* $i :: h :: t$ = *elems* $i :: l$ = $\{i\} \cup$ *elems* l. Tímto jsme dokázali první možný případ.

b) $i > h$

insert i l se evaluuje na $h::$*insert* i t. Označme si *insert* i t jako r. Jelikož platí *sorted* t a *length* $t = n$, můžeme použít indukční krok/náš předpoklad. Díky němu můžeme říct, že platí *sorted* r a *elems* r = *elems* $t \cup \{i\}$.

Pokud $t = []$, pak $l = [h]$ a *insert* i l se evaluuje na $h :: i$. Jelikož $h < i$, pak z první definice sorted platí $s_j \leq s_{j+1}$ pro každé přirozené číslo $j < n$, kde $n = 2$, tedy $j = 1$, $s_j = h$ a $s_{j+1} = i$. Seznam $h :: i$ je seřazený, tedy *insert* i l je seřazený.

Nechť t není prázdný seznam. Protože r je seřazený, platí $s_j \leq s_{j+1}$ pro každé přirozené číslo j, kde $1 < j < n + 2$. Protože $h < i$ a $h \leq t_k$ pro každé přirozené číslo $k \leq n$, pak platí $h \leq r_k$ pro každé $k \leq n + 1$, tedy i $h \leq r_1$, což v řeči seznamu s znamená $s_j \leq s_{j+1}$ pro $j = 1$. Tedy můžeme říct, že seznam s je seřazený, protože platí $s_j \leq s_{j+1}$ pro každé $j < n + 2$, tedy *insert* i l je seřazený.

Dále můžeme říct, že *insert* i l obsahuje prvky seznamu l a prvek i, protože *elems (insert i l)* = *elems (h :: insert i t)* = *elems* $h :: r$= $\{h\} \cup$ *elems* r = =$\{h\} \cup$ *elems* $t \cup \{i\}$ = $\{i\} \cup \{h\} \cup$ *elems* t = $\{i\} \cup$ *elems* l. Tímto jsme dokázali druhý možný případ.

Dokázali jsme, že platí *P(n+1)*, pokud platí *P(n)*. A jelikož jsme dokázali, že platí *P(0)* a že pokud *P(n)*, pak *P(n+1)*, tak jsme dokázali i *P(n)* pro pro každé $n \geq 0$. A tedy jsme dokončili celý důkaz korektnosti funkce *insert*.

3.2.2 Důkaz korektnosti funkce *sort*

Chceme dokázat, že pro libovolný seznam přirozených čísel l, který je seřazený, platí, že výsledný seznam *sort* l je také seřazený a obsahuje všechny prvky seznamu l.

Chceme tedy dokázat tvrzení $P(n)$ pro každé $n \geq 0$, kde

$P(n) = \forall$ seznam l : if length $l = n$ then sorted (sort l) $\wedge$ elems (sort l) = elems l

Základní bod indukce: $n = 0$
Chceme dokázat $P(0)$, tedy

$P(0) = \forall$ seznam l : if length $l = 0$ then sorted (sort l) $\wedge$ elems (sort l) = elems l

Takový seznam existuje právě jediný, a to prázdný seznam *[]*.
Nechť tedy l je prádný seznam []. Víme, že l má dělku 0.

Po evaluaci (vyhodnocení) *sort []* dostaneme *[]*.
Každý prázdný seznam je z definice *sorted* seřazený, takže i seznam *sort l* je seřazený.
Dále platí *elems (sort l) = elems (sort []) = elems [] = elems l*, neboli *sort []* obsahuje prvky seznamu [].
Tímto jsme dokázali základní krok indukce.

Induktivní krok: Nyní chceme dokázat, že pokud platí *P(n)*, pak platí i *P(n+1)*.
Předpokládejme, že *P(n)* platí pro libovolné pevné $n \geq 0$. Tedy, že platí

$P(n) = \forall$ seznam l : if length $l = n$ then sorted (sort l) $\wedge$ elems (sort l) = elems l

pro libovolné pevné $n \geq 0$.
Chceme dokázat, že platí *P(n+1)*, tedy

$P(n{+}1) = \forall$ seznam l : if length $l = n{+}1$ then sorted (sort l) $\wedge$ elems (sort l) = elems l

Nechť l je libovolný seznam přirozených čísel, pro který platí *length* l = *n+1*. Seznam l můžeme napsat jako $h :: t$, *length* $t = n$, a můžeme říct, že *elems* l = *elems* $t \cup \{h\}$.

Po evaluaci *sort* l dostaneme *insert* h *(sort* t*)*.
Protože t je seznam s délkou n, tak z předpokladu $P(n)$ platí, že *sort* t je seřazen a *elems (sort t) = elems t*.
Z důkazu korektnosti funkce *insert* víme, že pokud máme seřazený seznam r s délkou n a přirozené číslo i, pak seznam *insert* i r je seřazen a platí, že *elems (insert i r) = {i} ∪ elems r*.
Zvolme tedy r = *sort* t a $i = h$, pak *insert h (sort t)* je taky seřazen a platí, že *elems (sort l) = elems (insert h (sort t)) = {h} ∪ elems (sort t) = {h} ∪ elems t = elems l*.

Dokázali jsme, že platí *P(n+1)*, pokud platí *P(n)*. A jelikož jsme dokázali, že platí *P(0)* a že pokud *P(n)*, pak *P(n+1)*, tak jsme dokázali i *P(n)* pro pro každé $n \geq 0$. A tedy jsme dokončili celý důkaz korektnosti funkce *sort*.

A tímto jsme dokázali korektnost algoritmu *insertion sort*.

3.3 MERGE SORT - důkaz korektnosti algoritmu

Merge sort je řadící algoritmus, který používá strategii ”rozděl a panuj”. Jeho stabilní časová složitost je $O(n \log n)$, což z něj dělá efektivní algoritmus a tedy jasnou volbu při řazení velkých seznamů dat.

Algoritmus funguje tak, že neseřazený seznam rozdělí na dva menší seznamy, které budou seřazené, a ty následně sloučí do jednoho seřazeného seznamu. N začátku rekurzivně rozdělí původní seznam na dva menší seznamy, které rozdělí na dva menší seznamy, které zase rozdělí na dva menší seznamy, dokud nejsou seznamy jednoprvkové. Dva jednoprvkové seznamy sloučí do seřazeného dvouprvkového seznamu. Takhle postupně slučuje seřazené seznamy, dokud všechny menší seznamy nesloučí do jednoho výsledného seřazeného seznamu.

V našem případě, stejně jako u insertion sortu, použijeme rekurzivní pseudoimplementaci podobnou implementaci v jazyce Gallina, jelikož se podobá implementaci v poslední kapitole, kde v této podobě budeme s algoritmem pracovat [(UI20a, (UI20b].

```
Fixpoint split {X:Type} (l:list X) : (list X * list X) :=
  match l with
  | [] => ([],[])
  | [x] => ([x],[])
  | x1::x2::l' =>
    let (l1,l2) := split l' in
    (x1::l1,x2::l2)
  end.

Fixpoint merge l1 l2  {struct l1} :=
  let fix merge_aux l2 :=
  match l1, l2 with
  | [], _ => l2
  | _, [] => l1
  | a1::l1', a2::l2' =>
      if a1 <=? a2 then a1 :: merge l1' l2 else a2 :: merge_aux l2'
  end
  in merge_aux l2.

Function mergesort (l: list nat) {measure length l} :  list nat :=
  match l with
  | [] => []
  | [x] => [x]
  | _ => let (l1,l2) := split l in
         merge (mergesort l1) (mergesort l2)
  end.
```

Merge sort algoritmus používá pomocnou funkci *split*, která rozdělí seznam na dva menší seznamy tak podle vypočítené délky m.

Funkce *merge* sloučí dva seřazené seznamy tak, že výsledný seznam bude zase seřazený tím, že porovnává první prvky obou seznamů a postupně tvoří seřazený seznam.

Funkce *mergesort* seřadí vstupní seznam tím, že jej rozdělí na dva menší seznamy pomocí fnkce *split*, na které zavolá sebe sama, aby menší seznamy

zase rozdělila, dokud nebudou jednoprvkové. Poté na dvojice jednoprvkových seznamů zavolá funkci *merge*, která dva malé seznamy sloučí v jeden větší. Na dva větší seznamy, které byly vytvořeny pomocí funkce *merge*, se zase zavolá funkce *merge* a vytvoří jeden ještě větší seznam, dokud rekurze neprobublá zpět a my nedostaneme výsledný největší seřazený seznam.

Dokážeme, že se algoritmus chová korekně pro libovolný vstup, tedy že výsledný seznam bude seřazený a bude obsahovat všechny prvky jako seznam do algoritmu vstupující.

Důkaz rozdělíme na tři části, a to důkaz korektnosti funkce *split*, důkaz korektnosti funkce *merge* a důkaz korektnosti funkce *mergesort*. Dvě poslední funkce jsou rekurzivní, a tak se zase nabízí důkaz matematickou indukcí.

3.3.1 Důkaz korektnosti funkce *split*

Chceme dokázat, že pro libovolný seznam l s délkou n platí, že *elems l = elems (right (split l)) ∪ elems (left (split l))*.

Chceme dokázat tvrzení *P(n)* pro každé $n \geq 0$, kde

$$P(n) = \forall \text{ seznam } l :$$
$$\text{if } \text{ length } l = n$$
$$\text{then } \text{ elems (left (split } l)) \ \cup \ \text{elems (right (split } l)) = \text{elems } l$$

Nechť l je seznam délky $n \geq 0$. Pak podle algoritmu se $m = \lceil \frac{n}{2} \rceil$. Platí, že $m \leq n$ pro $n = \{0, 1\}$ a $m < n$ pro $n \geq 2$.

Pro $n = 0$ se $l = []$ a $m = 0$. A tedy $l_1 = [l_1, ..., l_0]$, tedy $l_1 = []$, a $l_2 = [l_0, ..., l_0]$, tedy také $l_2 = []$. Platí, že *elems (left (split l)) ∪ elems (right (split l)) = elems* l_1 *∪ elems* l_2 *= elems [] ∪ elems [] = ∅ ∪ ∅ = ∅ = elems [] = elems l*.

Pro $n = 1$ se $m = 1$ a $l = [l_1]$. A tedy $l_1 = [l_1, ..., l_1]$, tedy $l_1 = [l_1]$, a $l_2 = [l_2, ..., l_1]$, tedy $l_1 = []$. Platí, že *elems (left (split l)) ∪ elems (right (split l)) = elems* l_1 *∪ elems* l_2 *= elems* $[l_1]$ *∪ elems [] = elems* $[l_1]$ *∪ ∅ = elems* $[l_1]$ *= elems l*.

Pro $n \geq 2$ se $m = \lceil \frac{n}{2} \rceil$ a $l = [l_1, ..., l_n]$. A tedy $l_1 = [l_1, ..., l_m]$ a $l_2 = [l_{m+1}, ..., l_n]$. Seznamy l_1 a l_2 obsahují alespoň jeden prvek. Platí, že *elems (left (split l)) ∪ elems (right (split l)) = elems* l_1 *∪ elems* $l_2 = \{l_1, ..., l_m\} \cup \{l_{m+1}, ..., l_n\} = \{l_1, ..., l_n\} =$ *elems* $[l_1, \ ..., \ l_n]$ *= elems l*.

Tímto jsme dokázali platnost *P(n)* pro libovolné $n \geq 0$. A tedy jsme dokončili celý důkaz korektnosti funkce *split*.

3.3.2 Důkaz korektnosti funkce *merge*

Chceme dokázat, že pro dva libovolné seznamy přirozených čísel l_1 a l_2, které jsou oba seřazené a mají délky m a n, platí, že výsledný seznam *merge* l_1 l_2 je také seřazený a obsahuje všechny prvky seznamu l_1 a seznamu l_2.

Chceme dokázat tvrzení *P(n)* pro každé $n \geq 0$, kde

$P(m,n) = \forall$ seznam l_1, l_2 :
if length $l_1 = m \ \wedge$ length $l_2 = n \ \wedge$ sorted $l_1 \ \wedge$ sorted l_2
then sorted (merge $l_1 l_2$) $\wedge$ elems (merge $l_1 l_2$) = elems $l_1 \cup$ elems l_2

První základní krok indukce: pro $m = 0$ a pro libovolné n
Chceme tedy dokázat *P(0, n)*, kde

$P(0,n) = \forall$ seznam l_1, l_2 :
if length $l_1 = 0 \ \wedge$ length $l_2 = n \ \wedge$ sorted $l_1 \ \wedge$ sorted l_2
then sorted (merge $l_1 l_2$) $\wedge$ elems (merge $l_1 l_2$) = elems $l_1 \cup$ elems l_2

Takový seznam délky 0 existuje právě jediný, a to prázdný seznam [].
Nechť tedy l_1 je prázdný seznam [] délky 0 a l_2 je libovolný seřazený seznam délky n. Z definice *sorted* víme, že prázdný seznam je seřazený a tedy i l_1.

Po evaluaci *merge [] l_2* dostaneme l_2.
Protože jsme předpokládali, že l_2 je seřazen, je tedy seřazen i *merge [] l_2* a tedy i *merge l_1 l_2*.
Dále platí *elems (merge l_1 l_2) = elems (merge [] l_2) = elems l_2 = elems l_2* $\cup \emptyset$ *= elems l_2* $\cup$ *elems [] = elems l_2* $\cup$ *elems l_1*.
Tímto jsme dokázali první základní krok indukce *P(0, n)*.

Druhý základní krok indukce: pro libovolné m a pro $n = 0$
Chceme tedy dokázat *P(m, 0)*, kde

$P(m,0) = \forall$ seznam l_1, l_2 :
if length $l_1 = m \ \wedge$ length $l_2 = 0 \ \wedge$ sorted $l_1 \ \wedge$ sorted l_2
then sorted (merge $l_1 l_2$) $\wedge$ elems (merge $l_1 l_2$) = elems $l_1 \cup$ elems l_2

Takový seznam délky 0 existuje právě jediný, a to prázdný seznam [].
Nechť tedy l_2 je prázdný seznam [] délky 0 a l_1 je libovolný seřazený seznam délky m. Z definice *sorted* víme, že prázdný seznam je seřazený a tedy i l_2.

Po evaluaci *merge l_1 []* dostaneme l_1.
Protože jsme předpokládali, že l_1 je seřazen, je tedy seřazen i *merge l_1 []* a tedy i *merge l_1 l_2*.

Dále platí *elems (merge* l_1 l_2*) = elems (merge* l_1 *[]) = elems* l_1 *= elems* l_1 $\cup \emptyset =$ *elems* $l_1 \cup$ *elems [] = elems* $l_1 \cup$ *elems* l_2.

Tímto jsme dokázali druhý základní krok indukce *P(m, 0)*.

Induktivní krok: Nyní chceme dokázat, že pokud platí *P(m+1, n)* a *P(m, n+1)*, pak platí i *P(m+1, n+1)* pro libovolné pevné $m \geq 0$ a $n \geq 0$.

Předpokládejme, že *P(m+1, n)* a *P(m, n+1)* platí pro libovolné pevné $m \geq 0$ a $n \geq 0$. Tedy, že platí

$$
\begin{aligned}
&P(m+1, n) = \forall \text{ seznam } l_1, l_2 : \\
&\text{if } \text{ length } l_1 = m+1 \ \wedge \ \text{length } l_2 = n \ \wedge \ \text{sorted } l_1 \ \wedge \ \text{sorted } l_2 \\
&\text{then sorted } (\text{merge } l_1 l_2) \ \wedge \ \text{elems } (\text{merge } l_1 l_2) = \text{elems } l_1 \cup \text{elems } l_2
\end{aligned}
$$

a

$$
\begin{aligned}
&P(m, n+1) = \forall \text{ seznam } l_1, l_2 : \\
&\text{if } \text{ length } l_1 = m \ \wedge \ \text{length } l_2 = n+1 \ \wedge \ \text{sorted } l_1 \ \wedge \ \text{sorted } l_2 \\
&\text{then sorted } (\text{merge } l_1 l_2) \ \wedge \ \text{elems } (\text{merge } l_1 l_2) = \text{elems } l_1 \cup \text{elems } l_2
\end{aligned}
$$

pro libovolné pevné $m \geq 0$ a $n \geq 0$.

Chceme dokázat, že platí *P(m+1, n+1)*, kde

$$
\begin{aligned}
&P(m+1, n+1) = \forall \text{ seznam } l_1, l_2 : \\
&\text{if } \text{ length } l_1 = m+1 \ \wedge \ \text{length } l_2 = n+1 \ \wedge \ \text{sorted } l_1 \ \wedge \ \text{sorted } l_2 \\
&\text{then sorted } (\text{merge } l_1 l_2) \ \wedge \ \text{elems } (\text{merge } l_1 l_2) = \text{elems } l_1 \cup \text{elems } l_2
\end{aligned}
$$

Nechť l_1 je libovolný seřazený seznam přirozených čísel délky $m+1$ a l_2 je libovolný seřazený seznam přirozených čísel délky $n+1$. Seznam l_1 můžeme napsat jako $a_1 :: l_1'$, kde l_1' je seřazený seznam délky m. Seznam l_2 můžeme napsat jako $a_2 :: l_2'$, kde l_2' je seřazený seznam délky n.

Po evaluaci *merge* l_1 l_2 dostaneme dva možné výsputy podle toho, jak se vyhodnotí $a_1 \leq a_2$.

Poďme si tyto dva možné případy rozebrat.

Označme si *merge* l_1 l_2 jako l.

a) $a_1 \leq a_2$

merge l_1 l_2 se evaluuje na a_1*::merge* l_1' l_2. Protože l_1' je seřazený seznam délky m a l_2 je seřazený seznam délky $n+1$, z předpokladu *P(m, n+1)* platí, že *merge* l_1' l_2 je seřazen a *elems (merge* l_1' l_2*) = elems* $l_1' \cup l_2$. Označme si *merge* l_1' l_2 jako r. Víme, že délka r je $m+n+1$.

Protože r je seřazený, platí $s_j \leq s_{j+1}$ pro každé přirozené číslo j, kde $1 < j < m+n+2$, kde $m+n+2$ je délka seznamu s.

Protože platí $a_1 \leq a_2$ a $a_2 \leq l'_{2_k}$ pro každé přirozené číslo $k \leq n$, protože l_2 je seřazený, pak platí $a_1 \leq l_{2_k}$ pro každé přirozené číslo $k \leq n+1$. A protože zároveň platí $a_1 \leq l'_{1_k}$ pro každé přirozené číslo $k \leq m$, protože l_1 je seřazený, tak platí $a_1 \leq r_k$ pro každé přirozené číslo $k \leq m+n+1$. Tedy i $a_1 \leq r_1$, což v řeči seznamu s znamená $s_j \leq s_{j+1}$ pro $j = 1$. Tedy můžeme říct, že seznam s je seřazený, protože platí $s_j \leq s_{j+1}$ pro každé $j < m+n+2$, tedy *merge* l_1 l_2 je seřazený.

Dále můžeme říct, že *merge* l_1 l_2 obsahuje prvky seznamu l_1 a prvky seznamu l_2, protože *elems (merge* l_1 l_2*)* = *elems* $(a_1 :: merge\ l'_1\ l_2) = \{a_1\} \cup elems\ (merge\ l'_1\ l_2) = \{a_1\} \cup elems\ l'_1 \cup elems\ l_2 = elems\ l_1 \cup elems\ l_2$.
Tímto jsme dokázali první možný případ.

b) $a_2 < a_1$
merge l_1 l_2 se evaluuje na a_2*::merge* l_1 l'_2. Protože l'_2 je seřazený seznam délky n a l_1 je seřazený seznam délky $m+1$, z předpokladu *P(m+1, n)* platí, že *merge* l_1 l'_2 je seřazen a *elems (merge* l_1 l'_2*)* = *elems* $l_1 \cup l'_2$. Označme si *merge* l_1 l'_2 jako r. Víme, že délka r je $m+n+1$.
Protože r je seřazený, platí $s_j \leq s_{j+1}$ pro každé přirozené číslo j, kde $1 < j < m+n+2$, kde $m+n+2$ je délka seznamu s.
Protože platí $a_2 < a_1$ a $a_1 \leq l'_{1_k}$ pro každé přirozené číslo $k \leq m$, protože l_1 je seřazený, pak platí $a_2 \leq l_{1_k}$ pro každé přirozené číslo $k \leq m+1$. A protože zároveň platí $a_2 \leq l'_{2_k}$ pro každé přirozené číslo $k \leq n$, protože l_2 je seřazený, tak platí $a_2 \leq r_k$ pro každé přirozené číslo $k \leq m+n+1$. Tedy i $a_2 \leq r_1$, což v řeči seznamu s znamená $s_j \leq s_{j+1}$ pro $j = 1$. Tedy můžeme říct, že seznam s je seřazený, protože platí $s_j \leq s_{j+1}$ pro každé $j < m+n+2$, tedy *merge* l_1 l_2 je seřazený.

Dále můžeme říct, že *merge* l_1 l_2 obsahuje prvky seznamu l_1 a prvky seznamu l_2, protože *elems (merge* l_1 l_2*)* = *elems* $(a_2 :: merge\ l_1\ l'_2) = \{a_2\} \cup elems\ (merge\ l_1\ l'_2) = \{a_2\} \cup elems\ l_1 \cup elems\ l'_2 = elems\ l_1 \cup elems\ l_2$.
Tímto jsme dokázali druhý možný případ.

Dokázali jsme, že platí *P(m+1,n+1)*, pokud platí *P(m+1,n)* a *P(m,n+1)*. A jelikož jsme dokázali, že platí *P(0,n)*, *P(m,0)* a že pokud *P(m+1,n)* a *P(m,n+1)*, pak *P(m,n+1)*, tak jsme dokázali i *P(m,n)*. A tedy jsme dokončili celý důkaz korektnosti funkce *merge*.

3.3.3 Důkaz korektnosti funkce *mergesort*

Chceme dokázat, že pro libovolný seznam přirozených čísel l s délkou n platí, že výsledný seznam *mergesort* l je seřazený a obsahuje všechny prvky seznamu l.

Chceme dokázat tvrzení *P(n)* pro každé $n \geq 0$, kde

$P(n) = \forall$ seznam l :
if length $l = n$
then sorted (mergesort l) $\wedge$ elems (mergesort l) = elems l

První základní krok indukce: pro $n = 0$
Chceme tedy dokázat *P(0)*, kde

$P(0) = \forall$ seznam l :
if length $l = 0$
then sorted (mergesort l) $\wedge$ elems (mergesort l) = elems l

Takový seznam existuje právě jediný, a to prázdný seznam [].
Nechť tedy l je prázdný seznam [] délky 0.

Po evaluaci *mergesort []* dostaneme [].
Protože prázdný seznam je podle definice *sorted* seřazen, je tedy seřazen i *mergesort []* a tedy i *merge l*.
Dále platí *elems (mergesort l) = elems (merge []) = elems [] = elems l*.
Tímto jsme dokázali první základní krok indukce *P(0)*.

Druhý základní krok indukce: pro $n = 1$
Chceme tedy dokázat *P(1)*, kde

$P(1) = \forall$ seznam l :
if length $l = 1$
then sorted (mergesort l) $\wedge$ elems (mergesort l) = elems l

Nechť tedy $l = i$ je jednoprvkový seznam délky 1.

Po evaluaci *mergesort [i]* dostaneme $[i]$.
Protože každý jednoprvkový seznam je podle definice *sorted* seřazen, je tedy seřazen i *mergesort [i]* a tedy i *merge l*.
Dále platí *elems (mergesort l) = elems (mergesort [i]) = elems [i] = elems l*.
Tímto jsme dokázali druhý základní krok indukce *P(1)*.

Induktivní krok: Nyní chceme dokázat, že pokud platí *P(k)* pro každé k, kde $0 \leq k \leq n$, pak platí i *P(n+1)* pro libovolné pevné $n \geq 0$.
Předpokládejme, že *P(k)* platí pro všechna k, kde $1 \leq k \leq n$. Tedy, že platí

$P(k) = \forall$ seznam l :
if length $l = k$
then sorted (mergesort l) $\wedge$ elems (mergesort l) = elems l

pro každé k, kde $1 \leq k \leq n$.

Chceme dokázat, že platí *P(n+1)*, kde

$$P(n+1) = \forall \text{ seznam } l :$$
$$\text{if length } l = n+1$$
$$\text{then sorted (mergesort } l) \ \wedge \ \text{elems (mergesort } l) = \text{elems } l$$

kde $n \geq 1$.

Nechť l je libovolný seřazený seznam přirozených čísel délky $n+1$.

Po evaluaci *mergesort l* dostaneme *merge (mergesort (left (split l))) mergesort (right (split l))*.

Označme si *left (split l)* jako l_1, *right (split l)* jako l_2 a *merge (mergesort* l_1*)(mergesort* l_2*)* jako s.

Z důkazu funkce *split* můžeme o l_1 a l_2 říct, že *elems* $l_1 \cup$ *elems* l_2 = *elems l*. Dále můžeme říct, že délka l_1 je $\lceil \frac{n+1}{2} \rceil$ a délka l_2 je $n+1-(\lceil \frac{n+1}{2} \rceil+1)+1$, které označíme jako b a c. Platí, že $\lceil \frac{n+1}{2} \rceil < n+1$ a $n+1-(\lceil \frac{n+1}{2} \rceil+1)+1 < n+1$ pro $n \geq 1$.

Protože seznam l_1 má délku b, o které víme, že $b < n+1$, což můžeme přepsat jako $b \leq n$, tak z předpokladu *P(k)*, kde $k = b$, platí, že *mergesort* l_1 je seřazen a *elems (mergesort* l_1*) = elems* l_1. A protože seznam l_2 má délku c, o které víme, že $c < n+1$, což můžeme přepsat jako $c \leq n$, tak z předpokladu *P(k)*, kde $k = c$, platí, že *mergesort* l_2 je seřazen a *elems (mergesort* l_2*) = elems* l_2.

Z důkazu funkce *merge* víme, že pro každé dva seřazené seznamy k_1 a k_2 délek m_1 a m_2 platí, že *merge* k_1 k_2 je seřazen a že *elems (merge* k_1 k_2*) = elems* $k_1 \cup$ *elems* k_2. Zvolme tedy k_1 = *mergesort* l_1 a k_2 = *mergesort* l_2. Pak víme, že *merge* k_1 k_2 je seřazený, tedy že *merge (mergesort* l_1*) (mergesort* l_2*)* je seřazený, tedy že *merge (mergesort (left (split l))) mergesort (right (split l))* je seřazený.

Dále můžeme říct, že *mergesort l* obsahuje prvky seznamu l, protože *elems (mergesort l) = elems (merge (mergesort (left (split l))) mergesort (right (split l)))* = *elems (merge (mergesort* l_1*) (mergesort* l_2*)) = elems (mergesort* l_1*)* $\cup$ *elems (mergesort* l_2*) = elems* $l_1 \cup$ *elems* l_2 = *elems l*.

Dokázali jsme, že platí *P(n+1)*, pokud platí *P(k)* pro každé k, kde $1 \leq k \leq n$. A jelikož jsme dokázali, že platí *P(0)*, *P(1)* a že pokud *P(k)* pro každé k, kde $0 \leq k \leq n$, pak *P(n+1)*, tak jsme dokázali i *P(n)* pro pro každé $n \geq 0$. A tedy jsme dokončili celý důkaz korektnosti funkce *mergesort*.

A tímto jsme dokázali korektnost algoritmu *merge sort*.

Kapitola 4

Specifikace algoritmů

V této kapitole se zaměříme na koncept verifikovaných algoritmů a dva přístupy k jejich specifikaci. Verifikované programy jsou funkce, které vykonávají nějaký výpočet a zároveň obsahují důkaz o správnosti tohoto výpočtu vzhledem k dané specifikaci programu.

Popíšeme si dva hlavní přístupy verifikace těchto funkcí, slabou a silnou specifikaci. Poté se zaměříme na slabou specifikaci řadících algoritmů a její provedení v Coqu. Definujeme si dané specifikace v Coqu a ukážeme si pár důležitých knihovních funkcí, které budeme později používat [BC13].

Představme si relaci R typu $A \to B \to Prop$. Dále chceme vytvořit funkci f, která mapuje libovolný prvek a z A na prvek b z B tak, že bude platit $R\ a\ b$. Jak jsme již napověděli, tuto funkci můžeme definovat a dokázat dvěma způsoby tak, že bude splňovat danou specifikaci.

První způsob je definovat funkci f se slabou specifikací a následně dokázat potřebné teorémy a lemma. Tedy definujeme funkci f typu $A \to B$ s dokážeme lemma tvaru $\forall a : A,\ R\ a\ (f\ a)$.

Druhý způsob je definovat funkci f se silnou specifikací. Což znamená, že typ této funkce přímo říká, že vstupem je prvek a typu A a výstupem je prvek b typu B a důkaz, že platí $R\ a\ b$. V tomto přístupu říkáme, že vytváříme silně definovanou funkci, která váže vstupní prvek k výslednému. Tuto vlastnost ve slabé specifikaci dokazujeme dodatečně.

Nyní se podíváme na slabou specifikaci řadících algoritmů. V této práci budeme řadit seznamy přirozených čísel, a tedy obecná funkce řadícího algoritmu F bude typu $list\ nat \to list\ nat$, kde $list\ nat$ je typ seznam přirozených čísel, jak jsme si uváděli v kapitole Úvod do Coqu.

Od každé řadící funkce F chceme, aby výsledný seznam byl seřazený a aby obsahoval všechny prvky o stejném počtu výskytů jako seznam v argumentu funkce.

Vlastnost být seřazený v Coqu zadefinujeme následovně. Induktivní definice `sorted` říká, že seznam l je seřazen, pokud je prázdný, nebo pokud je jednoprvkový, nebo pokud první dva prvky x a y splňují relaci $x \leq y$ a seznam l bez prvního prvku x je také seřazen.

Druhou vlastnost, kterou jsme po řadící funkci požadovali, můžeme popsat

```
Inductive sorted : list nat -> Prop :=
| sorted_nil :
    sorted []
| sorted_1 : forall x,
    sorted [x]
| sorted_cons : forall x y l,
    x <= y -> sorted (y :: l) -> sorted (x :: y :: l).
```

knihovním typem `Permutation`. Je to vlastnost dvou seznamů, která je popsána čtyřmi konstruktory. Jeden seznam je permutací druhého, pokud jsou oba prázdné. Pokud dva seznamy splňují vlastnost `Permutation`, pak budou tuto vlastnost splňovat i dva nové seznamy, které vznikli přídáním prvku `x` na začátek těchno seznamů. Dva seznamy splňují vlastnost `Permutation`, pokud jsou stejné až na prohození prvních dvou prvků. A na závěr, pokud jsou ve vztahu `Permutation` jeden a druhý seznam a zároveň druhý a třetí seznam, pak jsou ve vztahu `Permutation` i první a třetí seznam.

Po krátkém pozorování vidíme, že tato definice je úplná a že vlastnost `Permutation` je opravdu permutace, jakou známe, a splňuje všechny vlastnosti, které bychom od permutce očekávali.

```
Inductive Permutation {A : Type} : list A -> list A -> Prop :=
  | perm_nil : Permutation [] []
  | perm_skip : forall (x : A) (l l' : list A),
                Permutation l l' ->
                Permutation (x :: l) (x :: l')
  | perm_swap : forall (x y : A) (l : list A),
                Permutation (y :: x :: l) (x :: y :: l)
  | perm_trans : forall l l' l'' : list A,
                 Permutation l l' ->
                 Permutation l' l'' -> Permutation l l''.
```

Věty o výše definovaných vastnostech, které požadujeme po řadící funkci F, v Coqu definujeme takto.

```
Theorem F_sorted: forall l, sorted (F l).

Theorem F_perm: forall l, Permutation l (F l).
```

Po vytvoření důkazů těchto dvou vět zadefinujeme výsledný teorém, který shrnuje všechny požadavky na funkci F, který dokážeme aplikací těchto dvou vět. Tímto jsme dokázali korenktnost řadící funkce F se slabou specifikací. Takový teorém obecně zadefinujeme pro libovolnou funkci F. Pokud tento teorém dokážeme, ukážeme, že F je řadící algoritmus korektně definovaný se slabou specifikací.

```
Definition is_a_sorting_algorithm (f: list nat -> list nat) :=
    forall al, Permutation al (f al) /\ sorted (f al).
```

V následující kapitole budeme používat knihovní lemmata týkající se permutací a seznamů k dokázaní korektnosti řadících algoritmů. Pojďme si ukázat definice několika z nich a jejich použití na jednoduchém příkladu.

Pokud si v důkazu nemůžeme vzpomenou na název knihovního lemmatu, které potřebujeme, můžeme použít klíčové slovo `Search` a dostaneme všechna lemmata týkající se hledaného slova.

```
Search Permutation.
```

Zde můžeme vidět hlavičky knihovních lemmat, která budeme používat v důkazech korektnosti řadících algoritmů.

```
Permutation_refl: forall [A : Type] (l : list A), Permutation l l

Permutation_app_comm:
  forall [A : Type] (l l' : list A), Permutation (l ++ l') (l' ++ l)

Permutation_nil:
  forall [A : Type] [l : list A], Permutation [] l -> l = []

Permutation_nil_cons:
  forall [A : Type] [l : list A] [x : A], ~ Permutation [] (x :: l)

Permutation_cons_inv:
  forall [A : Type] [l l' : list A] [a : A],
  Permutation (a :: l) (a :: l') -> Permutation l l'

Permutation_app_head:
  forall [A : Type] (l : list A) [tl tl' : list A],
  Permutation tl tl' -> Permutation (l ++ tl) (l ++ tl')

Permutation_app:
  forall [A : Type] [l m l' m' : list A],
  Permutation l l' ->
  Permutation m m' -> Permutation (l ++ m) (l' ++ m')

app_nil_r
     : forall (A : Type) (l : list A), l ++ [] = l
```

A zde můžeme vidět jejich použití. Kromě těchto lemmat budeme používat i konstruktory `Permutation` `perm_skip`, `perm_swap` a `perm_trans`, jak je vidět v příkladu.

```
Example permut_example: forall (a b: list nat),
  Permutation (5 :: 6 :: a ++ b) ((5 :: b) ++ (6 :: a ++ [])).
Proof.
intros.
change (5 :: 6 :: a) with ([5] ++ [6] ++ a).
simpl.
apply perm_skip.
apply perm_trans with (b ++ 6 :: a).
- replace (6 :: a ++ b) with ((6 :: a) ++ b).
  + apply Permutation_app_comm.
  + apply app_comm_cons.
- apply Permutation_app_head.
  apply perm_skip.
  replace (a ++ []) with (a).
  + apply Permutation_refl.
  + symmetry. apply app_nil_r.
Qed.
```

Kapitola 5

Korektnost Insertion sortu a Merge sortu v Coqu

V předchozích kapitolách jsme se naučili dokazovat v Coqu pomocí taktik, zavedli jsme slabou specifikaci pro algoritmy a konkrétní specifikaci pro řadící algoritmy. Zadefinovali jsme vlastnosti v Coqu, které od řadících algoritmů požadujeme a ukázali si lemmata, která použijeme v důkazu jejich korektnosti. Dokázali jsme korektnost Insertion sortu a Merge sortu klasicky. Nyní můžeme všechny tyto znalosti využít a dokázat korektnost zmíněných dvou algoritmů v Coqu a ukázat rozdílný přístup oproti klasickému dokazování.

Definujeme si taktitku `bdestruct` složenou z více taktik. Tato taktika funguje jako `destruct`, který jsme si definovali v předešlé kapitole. Navíc nahradí veškeré výskyty destruovaného výrazu za výraz aktuální, podle větve `destructu`. Také nahradí nerovnost v destructovaném výrazu za booleovskou proměnnou podle větve `destructu`.

Taktika `inv` je obdobně definovaná, jako taktika `bdestruct`. Je definovaná na bázi taktiky `inversion`, která převádí předpoklady do základních tvarů. Taktika `inv` navíc vymaže z předpokladů ty, ketré jsou redundantní.

5.1 Insertion sort

Nejprve naimplementujme algoritmus Insertion sort. Použijeme implementaci z knihy Software Foundantions, volume 3: verified functiobal algorithms. Pokud bysme chtěli naimplementovat algoritmus sami, skončili bychom s indentickou implementací. Logika jazyka Gallina by nás dovedla ke stejnému cíli [App23, BC13].

Nyní chceme dokázat, že Insertion sort je korektně definován. Tedy chceme dokázat toto tvrzení.

K důkazu tohoto tvrzení budeme potřebovat dva důkazy, a to důkaz, že `sort` vrací seřazený seznam, a důkaz, že výsledný seznam je permutací seznamu do funkce vstupujícího.

Pojďme ale začít popořadě.

```
Fixpoint insert (i : nat) (l : list nat) :=
  match l with
  | [] => [i]
  | h :: t => if i <=? h then i :: h :: t else h :: insert i t
  end.

Fixpoint sort (l : list nat) : list nat :=
  match l with
  | [] => []
  | h :: t => insert h (sort t)
  end.

Theorem insertion_sort_correct:
    is_a_sorting_algorithm sort.
```

Nejprve dokažme větu, která říká, že pro každý prvek a a seřazený seznam l platí, že seznam `sorted (insert a l)` je taky seřazen.

```
Lemma insert_sorted:
  forall a l, sorted l -> sorted (insert a l).
```

Tuto větu dokážeme indukcí podle předpokladu `sorted l`, čímž vzniknou tři cíle. První cíl je triviální a v dalších dvou použijeme taktiku `bdestruct` na nerovnosti prvků kvůli tomu, jak je `sorted` definován. Poté aplikujeme konstruktory `sorted`. Místo `apply sorted_cons` můžeme použít taktiku `constructor`. Tato taktika aplikuje konstruktor stejně tak, jako kdybychom napsali `apply sorted_cons`. Rozdíl je v tom, že `constructor` dokáže najít vhodný konstruktor a my si nemusíme pamatovat názvy každého z nich.

Dále dokážeme stejné tvrzení, ale pro funkci `sort`. Důkaz tohoto tvrzení bude kratší, protože již máme dokázáno předešlou větu, která nám velmi pomůže.

```
Theorem sort_sorted: forall l, sorted (sort l).
```

Tuto větu dokážeme indukcí přes l, v první větvi použijeme konstruktor a ve druhé předešlou větu a přepoklad.

Nyní dokážeme větu, která mluví o permutacích a funkci `insert`.

```
Lemma insert_perm: forall x l,
    Permutation (x :: l) (insert x l).
```

Tuto větu dokážeme také indukcí přes l. V první větvi důkazu použijeme lemma o permutacích zmiňované v předešlé kapitole. Ve druhé použijeme `bdestruct` kvůli implementaci `sorted`, čímž se nám druhá větev rozpadne na dvě. V obou použijeme lemmata o permutacích, konkrétně `perm_trans`.

S tímto dokázaným tvrzením v rukávu se nám bude dokazovat tvrzení o `sort` a permutacích opět jednodušeji.

```
Lemma insert_perm: forall x l,
    Permutation (x :: l) (insert x l).
```

Tuto větu dokážeme opět indukcí přes seznam l a podobně jako v předešlém důkazu použijeme `perm_trans` a předešlou větu.

Jelikož jsme dokázali obě tvrzení o funkci `sort`, použijeme tyto dvě tvrzení k důkazu teorému `insertion_sort_correct`, který jsme zmínili výše. Použijeme taktiku `unfold` a aplikujeme obě dokázaná tvrzení. Tímto jsme dokázali teorém a tedy i korektnost funkce `sort` a tedy i korektnost algoritmu Insertion sort se zmíněnou implementací.

5.2 Merge sort

Stejně jako u Insertion sortu, i zde začneme implementací algoritmu a použijeme tu z knihy Software Foundantions, volume 3: verified functiobal algorithms. I zde, pokud bychom se pokoušeli o vlastní implementaci, dobrali bychom se podobné, ne-li stejné [App23, BC13].

```
Fixpoint split {X:Type} (l:list X) : (list X * list X) :=
  match l with
  | [] => ([],[])
  | [x] => ([x],[])
  | x1::x2::l' =>
    let (l1,l2) := split l' in
    (x1::l1,x2::l2)
  end.

Fixpoint merge l1 l2  {struct l1} :=
  let fix merge_aux l2 :=
  match l1, l2 with
  | [], _ => l2
  | _, [] => l1
  | a1::l1', a2::l2' =>
      if a1 <=? a2 then a1 :: merge l1' l2 else a2 :: merge_aux l2'
  end
  in merge_aux l2.

Function mergesort (l: list nat) {measure length l} :  list nat :=
  match l with
  | [] => []
  | [x] => [x]
  | _ => let (l1,l2) := split l in
         merge (mergesort l1) (mergesort l2)
  end.
```

Všimněme si, že funkce `mergesort` začíná klíčovým slovem `Function`. Tohle klíčové slovo v sobě ukrývá více funkcionalit než klíčové slovo `Fixpoint`. Rekurzivní funkce mohou být v Coqu zadefinovány pouze tehdy, pokud Coq rozklíčuje, že argument, přes který se rekurze provádí, se s každým voláním

funkce strukturálně zmenšuje, a tedy, že funkce někdy skončí, terminuje. To se Coqu v naší definici `mergesort` nepodaří a musíme použít slovo `Function`, které nám umožňuje Coqu specifikovat, že l se strukturálně zmenšuje. Coq nám uvěří, ale požaduje po nás tohle tvrzení dokázat. K tomu ale potřebujeme lemma `split_len`.

K tomu, abychom lemma `split_len` dokázali, musíme učinit pár kroků. Zadefinujme si nový indukční princip.

```
Definition list_ind2_principle:=
    forall (A : Type) (P : list A -> Prop),
      P [] ->
      (forall (a:A), P [a]) ->
      (forall (a b : A) (l : list A), P l -> P (a :: b :: l)) ->
      forall l : list A, P l.
```

S tímto indukčním principem si zadefinujeme nové lemma a dokážeme ho.

```
Lemma split_len': list_ind2_principle ->
    forall {X} (l:list X) (l1 l2: list X),
    split l = (l1,l2) ->
    length l1 <= length l /\
    length l2 <= length l.
```

Použijeme `unfold` na nově definovaný indukční princip a použijeme ho v indukci podle seznamu l.

Lemma `split_len'` se nám podařilo dokázat, ale musíme ještě dokázat již zadefinovaný indukční princip, dokážeme ho definicí funkce `list_ind2`, která popisuje, jak se má indukční princip provádět.

S tímto důkazem můžeme konečně zadefinovat a dokázat lemma `split_len`.

```
Lemma split_len: forall {X} (l:list X) (l1 l2: list X),
    split l = (l1,l2) ->
    length l1 <= length l /\
    length l2 <= length l.
```

Dokážeme ho aplikací lemmatu `split_len'` a funkce indukčního principu `list_ind2`.

Nyní použijeme dokázané lemma `split_len` v důkazu terminace funkce `mergesort`. Důkaz sestává ze dvou částí, a to z důkazu, ve kterém ukazujeme, že seznam `l1`, vzniklý z `split l`, je kratší než původní seznam `l`, a z analogického důkazu, ve kterém ukazujeme, že seznam `l2`, vzniklý z `split l`, je kratší než původní seznam `l`.

Teď když jsme správně zadefinovali všechny funkce algoritmu Merge sort, můžeme dokázat, že je algoritmus s naší implementací korektně definován. K tomu potřebujeme dokázat toto tvrzení.

```
Theorem mergesort_correct:
  is_a_sorting_algorithm mergesort.
```

Začněme důkazem lemmatu `sorted_merge1`, které budeme potřebovat v následujícím důkazu.

```
Lemma sorted_merge1 : forall x x1 l1 x2 l2,
    x <= x1 -> x <= x2 ->
    sorted (merge (x1::l1) (x2::l2)) ->
    sorted (x :: merge (x1::l1) (x2::l2)).
```

Tohle lemma říká, že pokud je prvek `x` menší roven oběma prvím prvkům v seznamech, ketré slučujeme a sloučení těchto dvou seznamů je seřazený seznam, pak i spojení onoho prvku `x` a seřazeného seznamu je seřazený seznam. Dokážeme ho použitím `bdestruct`.

Nyní dokázané lemma použijeme v důkazu, že funkce `merge` vrátí seřazený seznam, pokud jeho dva vstupní agrumenty byly také seřazené seznamy.

```
Lemma sorted_merge : forall l1, sorted l1 ->
                     forall l2, sorted l2 ->
                     sorted (merge l1 l2).
```

Tento důkaz je jedním z nejsložitějších, které v této práci budeme předádět. Lemma dokážeme indukcí na seznamu $l1$, kde v každé větvi indukce použijeme indukci na seznamu $l2$. V každé větvi pak použijeme `bdestruct` kvůli definici `sorted`. V jedné z větví použijeme taktiku `inv` a lemma `sorted_merge1`. V další větvi používáme knihovní lemmata `leb_correct` a `leb_correct_conv`, která převádějí výrok s nerovností na rovnost mezi booleanem a funkcí vracející boolean.

Dále dokážeme lemma `mergesort_sorts`, které říká, že `mergesort` vrací seřazený seznam. Lemma dokážeme indukcí na l s indukčním principem,

```
Lemma mergesort_sorts: forall l, sorted (mergesort l).
```

který je vygeneroval při definici funkce `mergesort`. Použijeme předešlé lemma `sorted_merge`.

Nyní dokážeme lemma týkající se permutací a funkce `split`. Lemma říká, že argument funkce `split` je permutací spojení seznamů, které jsou jejími výstupy. Lemma dokážeme indukcí podle l za použití indukčního principu

```
Lemma split_perm : forall {X:Type} (l l1 l2: list X),
    split l = (l1,l2) -> Permutation l (l1 ++ l2).
```

`list_ind2`. V každé větvi použijeme taktiku `inv` a knihovní lemmata o permutacích. V poslední větvi použijeme ještě `destruct` na předpoklad a použijeme mnoho knihovních lemmat o permutacích.

Právě dokázané lemma použijeme v následujícím důkazu lemmatu `merge_perm`, které analogicky říká, že výsledný seznam funkce `merge` je permutací spojení vstupních seznamů. Důkaz opět začneme indukcí před $l1$ a v každé větvi

```
Lemma merge_perm: forall (l1 l2: list nat),
    Permutation (l1 ++ l2) (merge l1 l2).
```

použijeme indukci přes $l2$. Použijeme knihovní lemmata o permutacích a o spojování seznamů. Použijeme `bdestruct` a opět spoustu knihovních lemmat.

V posledním lemmatu `mergesort_perm` říkajícím, že vstupní seznam je permutací výstupního seznamu z funkce `mergesort`. Důkaz začneme indukcí

```
Lemma mergesort_perm: forall l, Permutation l (mergesort l).
```

na l za použití vygenerovaného indukčního principu `mergesort_ind`. Opět použijeme spoustu knihovních lemmat o permutacích.

Jelikož jsme dokázali obě tvrzení o funkci `mergesort`, použijeme tyto dvě tvrzení k důkazu teorému `insertion_sort_correct`, který jsme zmínili výše. Použijeme taktiku `unfold` a aplikujeme obě dokázaná tvrzení. Tímto jsme dokázali teorém a tedy i korektnost funkce `mergesort` a tedy i korektnost algoritmu Merge sort se zmíněnou implementací.

Kapitola 6

Závěr

Tato bakalářská práce se zaměřila na problematiku formálního dokazování korektnosti softwaru. Při vývoji softwaru je klíčové, aby software byl nejen funkční, ale také bylo korektně definované jeho chování. To je důležité u spolehlivosti a bezpečnosti softwarových systémů. Práce se konkrétně zaměřila na využití důkazového asistenta Coq a jazyka Gallina, které umožňují specifikovat chování programů a formálně dokazovat jejich korektnost.

Během práce jsme se naučili pracovat s Coqem, jeho jazykem a taktikami, a seznámili jsme se základní metody specifikace programu, a to slabou a silnou specifikaci. Tyto znalosti jsme aplikovali na dva konkrétní algoritmy: Insertion sort a Merge sort. Oba algoritmy byly implementovány v Coqu a jejich korektnost byla formálně dokázána pomocí tohoto nástroje. Současně jsme provedli formální důkazy korektnosti těchto algoritmů klasicky, což nám umožnilo porovnat rozdílné přístupy v dokazování korektnosti.

Výsledky této práce ukazují, že použití důkazového asistenta Coq verifikuje algoritmy bezchybně. Naproti tomu klasická formální verifikace není jasně definovaná, protože ji dokazuje člověk svými myšlenkami. Často se stává, že spousta tvrzeních se formálně nedokazuje, protože jsou zřejmá. Tato zřejmá tvrzení nám Coq v důkazech neodpustí.

Přestože jsou formální důkazy často náročné na čas a vyžadují hluboké porozumění matematických konceptů, výsledkem nám jen bezchybný formální důkaz korektnosti.

Celkově tato práce přispěla k lepšímu porozumění formálních metod ve vývoji softwaru a ukázala praktické aplikace těchto metod na konkrétních algoritmech.

V budoucnu bychom mohli na tuto práci navázat a rozšířit ji o důkazy korektnosti se silnou specifikací a ukázat tuto metodu na složitějších algotitmech nebo datových strukturách, jako jsou vyhledávací stromy a BFS A DFS algoritmy.

Literatura

[App23] Andrew W. Appel, *Verified functional algorithms*, 2023.

[BC13] Yves Bertot and Pierre Castéran, *Interactive theorem proving and program development: Coq'art: the calculus of inductive constructions*, Springer Science and Business Media, 2013.

[CP24] Benjamin C. Pierce, *Software foundations: Logical foundations*, Logical foundations (2024).

[(UI20a] Har-Peled (UIUC), *10.6 merge sort*, 2020.

[(UI20b] ———, *10.6 merge sort*, 2020.

[Uni08] Cornell University, *Cs 3110 recitation 11: Proving correctness by induction*, Recitation 11: Proving Correctness by Induction (2008).

[CP24, App23, BC13, (UI20a, (UI20b, Uni08]