3. Algoritmien oikeellisuus

(Harel luku 5)

"Opettakaa minulle, missä olen mennyt harhaan"
[Job 6:24]

• Kuinka varmistua siitä, että algoritmi toimii oikein?

Ohjelmointia harjoittaneet tietävät, että oikein toimivien ohjelmien aikaansaaminen on haastavaa. Miljoonavahinkoja aiheuttaneista ja jopa ihmishenkiä vaatineista ohjelmistovirheistä kerrotaan tosia tarinoita.

Tietokoneohjelmien oikeellisuutta pyritään normaalisti osoittamaan **testaamalla** eli kokeilemalla toimintaa edustavalla ja kattavalla joukolla syötteitä.

Mahdollisia syötteitä usein ääretön määrä $\leadsto$ testaamalla ei voi saada oikeellisuudesta *varmuutta*.

Meidän pitää voida varmistua siitä, että esittämämme yleinen ratkaisumenetelmä, algoritmi, toimii oikein.

Algoritmeja (ja kriittisiä ohjelmia) voidaan **verifioida** eli **todistaa oikeaksi**.

Ohjelmien verifiointia käsitellään perusteellisemmin kurssilla *Algoritmien oikeellisuus ja johtaminen*.

Mitä verifioinnissa on osoitettava?

Algoritmin pitää olla ongelman määrityksen mukainen ratkaisu: Sen on tuotettava kaikilla sallituilla syötteillä haluttu, spesifikaation mukainen tulos.

Algoritmi on *virheellinen*, *jos* jollain sallitulla syötteellä

- saadaan väärä vastaus tai
- suoritus voi joutua virheelliseen tai määrittelemättömään tilanteeseen (nollalla jako, yritys edetä puussa lehtisolmun lapseen, viittaus taulukon ulkopuolelle ym.) tai
- suoritus on päättymätön, ns. **ikuinen silmukka**.

Osittainen ja täysi oikeellisuus

Algoritmi tai ohjelma on **osittain oikeellinen**, jos se tuottaa sallituista syötteistä oikean tuloksen aina, *mikäli suoritus päättyy*.

Osittain oikeellinen algoritmi siis ei anna väärää vastausta, mutta voi joutua joskus ikuiseen silmukkaan.

Sanomme, että algoritmi/ohjelma **terminoi**, jos sen suoritus päättyy millä tahansa ongelman määrityksen mukaisella syötteellä.

Osittain oikeellinen ja lisäksi terminoiva algoritmi/ohjelma on **täysin oikeellinen**

Ks. Harel, kuva 5.3.

Osittaisen oikeellisuuden osoittaminen

Osittaisen oikeellisuuden osoittamisessa näytetään, että algoritmin suorituksessa *ei tapahdu vääriä asioita*:

• algoritmi ei tuota vääriä tuloksia ja

• muuttujilla ei ole virhetilanteeseen johtavia arvoja (esim. nolla jakajana tai indeksi taulukon rajojen ulkopuolella)

Argumentointi tapahtuu liittämällä algoritmissa sopiviin varmistuskohtiin (checkpoint) **väittämiä** (assertion). Väittämillä ilmaistaan asiantila, jonka halutaan osoittaa olevan k.o. kohdassa voimassa.

Yleensä väittämissä käytetään algoritmin muuttujia:

Esim. "taulukon alkiot 1, ..., I on järjestetty."

Algoritmin **alkuväittämä** (initial assertion) on yleensä sallitut syötteet kuvaava ehto. Vastaavasti **loppuväittämä** (final assertion) kuvaa sen, että tulosteen ja syötteen suhde on ongelman ratkaisulta haluttu.

Harel, kuva 5.5

Osittainen oikeellisuus perustellaan osoittamalla, että algoritmiin sijoitetut väittämät ovat **invariantteja** eli tosia aina kun suoritus saavuttaa niiden sijainnin.

Invarianttien verifiointi tapahtuu osoittamalla, että mikään siirtymä suorituksessa uuteen varmistuskohtaan ei tee siihen liittyvää väittämää epätodeksi.

Ns. Floydin menetelmä (Robert Floyd 1967)

Esim. Merkkijonon peilikuva

Toteutettava merkkijonon kääntävä funktio **reverse**, jolla reverse(" $c_1 \ldots c_n$ ") = " $c_n \ldots c_1$ ".

Esim. reverse("kalakauppias") = "saippuakalak"

Merkitään tyhjää merkkijonoa symbolilla λ. Oletetaan merkkijonojen käsittelyyn funktiot **head**(" $c_1 \ldots c_n$ ") = " c_1 "ja **tail**(" $c_1 \ldots c_n$ ") = " $c_2 \ldots c_n$ ".

Esim. head("kalakauppias") = "k"ja tail("kalakauppias") = "alakauppias".

Merkitään lisäksi merkkijonojen katenointia (Java-tyyliin) plus-merkillä:
"kala" + "kauppias" = "kalakauppias".

Funktio reverse(S) voidaan nyt toteuttaa allaolevalla algoritmilla (Harel kuva 5.6):

```
X:= S;
Y:= λ;
while X ≠ λ do
        Y:= head(X) + Y;
        X:= tail(X);
od;
output Y;
```

Intuitiivinen idea: Jonon S alusta siirretään yksitellen merkkejä alunperin tyhjän jonon Y alkuun; Y ”kasvaa oikealta vasemmalle”

Osoitetaan ensin algoritmin osittainen oikeellisuus. Lisätään algoritmiin (kommentteina) kolme väittämää (Harel kuva 5.7):

// I: S on merkkijono
X:= S;
Y:= λ;
while X $\neq \lambda$ **do**// II: S = reverse(Y) + X
    Y:= head(X) + Y;
    X:= tail(X);
od;
// III: Y = reverse(S)
output Y;

Väittämä II kuuluu suorituksessa siihen kohtaan jossa silmukkaehtoa testataan.

Huom: väittämät I ja III muodostavat reverse-ongelman spesifikaation.

Osoitetaan väittämien invarianssi tarkistamalla kaikki mahdolliset varmistuskohtien väliset siirtymät (I $\rightarrow$ III, I $\rightarrow$ II, II $\rightarrow$ II ja II $\rightarrow$ III)

Mahdolliset suoritukset ovat I $\rightarrow$ III ja
I $\rightarrow$ II $\rightarrow$ [II $\rightarrow \cdots$ II $\rightarrow$] III (Harel kuva 5.8);
siten yksittäisten siirtymien verifiointi
osoittaa, että algoritmin lopussa sen
loppuväittämä on tosi.

Huom: varmistuskohdat valittu siten, että
niiden välisiin siirtymiin ei sisälly silmukoiden
toistoja

Hahmotellaan siirtymiin liittyvien väittämien
invarianssitodistus:

I $\rightarrow$ III: **while**-silmukkaa ei suoriteta lainkaan,
joten X=S=λ, ja siis pisteessä III
Y=λ=reverse(S)

I $\rightarrow$ II: väite II pätee, koska X=S ja
Y=λ=reverse(Y)

II $\rightarrow$ II: Merkitään muuttujien arvoja ennen silmukan rungon suoritusta X_1 ja Y_1 sekä sen jälkeen X_2 ja Y_2. Ennen siirtymää siis S = reverse(Y_1) + X_1. Jos S = $c_1 \ldots c_n$, arvot ovat siis siirtymän alkuehdon mukaan jollain $1 \leq k \leq n$ seuraavat:

$$\overbrace{c_{k-1} \cdots c_1}^{Y_1} \; \overbrace{c_k \cdots c_n}^{X_1}$$

Silmukan rungon suorituksen ansiosta Y_2 = head(X_1) + Y_1 ja X_2 = tail(X_1). Siirtymän jälkiehdon II, S = reverse(Y_2) + X_2, nähdään siis olevan voimassa:

$$\overbrace{c_k \cdots c_1}^{Y_2} \; \overbrace{c_{k+1} \cdots c_n}^{X_2}$$

Siirtymä II $\rightarrow$ III tapahtuu kun X=λ, jolloin siis S=reverse(Y) eli Y=reverse(S).

Olemme osoittaneet algoritmin osittaisen oikeellisuuden.

Terminoinnin osoittaminen

Algoritmin terminointi voidaan osoittaa valitsemalla sopiva suorituksen edetessä pienenevä suure, ns. **konvergentti**, joka kuitenkaan ei voi pienetä loputtomasti. Kuten väittämät, konvergentti riippuu usein algoritmin muuttujista tai tietorakenteista.

Esim. käsittelemättömien tietueiden lkm, lajittelualgoritmissa väärässä kohtaa jonoa olevien alkioiden lkm tms.

Jatketaan reverse-algoritmin parissa osoittamalla sen terminointi.

Reverse-algoritmin suoritus voi olla päättymätön vain jos varmistuskohta II ohitetaan äärettömän monesti. Osoitetaan tämä mahdottomaksi valitsemalla konvergentti, joka pienenee jokaisella II-kohdan ohituksella.

Merkkijonon X pituus selvästi pienenee jokaisessa silmukan toistossa. Toisaalta silmukan suoritus päättyy, kun X:n pituus $= 0$.

$\rightsquigarrow$ algoritmi terminoi.

Olemme osoittaneet algoritmin täyden oikeellisuuden.

Automaattinen verifiointi?

Algoritmien huolellinen verifiointi käsin on selvästikin työlästä.

Missä määrin tietokone voi auttaa verifioinnissa?

Voiko algoritmien verifioimisen suorittaa algoritmisesti?

Harel, kuva 5.4

Valitettavasti verifioinnin täydellinen automatisointi ei ole mahdollista.

Varmistuskohtien valinta siten, että jokainen toistorakenteen suoritus kulkee jonkin varmistuskohdan kautta voidaan automatisoida.

Väittämien ja konvergenttien valintaa ja niihin liittyviä todistuksia ei voida suorittaa algoritmisesti; sivuamme tätä myöhemmin.

Ohjelmien oikeaksitodistamista tukevia **todistusjärjestelmiä** tutkitaan ja kehitetään. Ne toimivat vuorovaikutuksessa käyttäjän kanssa ja pystyvät huolehtimaan suurelta osin väittämien verifiointiin tarvittavasta työläästä kaavamanipuloinnista.

Huom: väittämiä voi käyttää myös ohjelmien testauksen apuna: Esim. C-kielen **assert**-makro, joka tuottaa virheilmoituksen jos sen argumenttina oleva ehto on suorituksessa epätosi.

Rekursiivisen algoritmin verifiointi

Esim. Hanoin tornit

```
Moves(N,X,Y,Z):
// X on lähtö-, Y kohde- ja Z apupylväs
    if N=1 then
        output "siirrä" + X + "→" + Y;
    else
        Moves(N-1, X, Z, Y);
        output "siirrä" + X + "→" + Y;
        Moves(N-1, Z, Y, X);
    fi;
```

Rekursiivisten algoritmien verifiointi voi olla suhteellisen helppoa.

Terminointi: Palautetaan mieliin Moves-proseduurin (binäärinen) kutsupuu. Suoritus on päättymätön vain jos kutsupuu on ääretön. Parametri N pienenee jokaisessa rekursiivisessa kutsussa yhdellä

$\leadsto$ Moves(N, X, Y, Z)-suorituksen kutsupuun syvyys on N, kun N on positiivinen kokonaisluku. Algoritmi siis terminoi.

Osittainen oikeellisuus voidaan osoittaa induktiolla. Osoitetaan, että kaikilla $N \geq 1$ pätee $P(N)$:

"Kutsun Moves(N, X, Y, Z) suoritus tuottaa jonon sallittuja siirtoja, joilla (sallitussa tilanteessa) N pienintä kiekkoa voidaan siirtää sauvasta X (mikäli ne ovat siinä) sauvaan Y eikä muihin kiekkoihin kosketa."

Tapaus $N = 1$ on selvä: "siirrä X $\rightarrow$ Y" on sallittu yhden kiekon siirto maalipylvääseen, joka koskee ainoastaan pienimpään kiekkoon.

Induktioaskel $N > 1$: ks. Harel kuva 5.10 selityksineen.

Terminointi + osittainen oikeellisuus $\rightsquigarrow$ täysi oikeellisuus.

Huom: usein algoritmin verifiointi jo rinnan sen suunnittelun kanssa on hyödyllistä.
(ns. **as-you-go** verification;
esimerkkejä mahdollisesti harjoitustehtävinä.)

Logiikka ja laskenta

(VAROITUS: Loput tästä luvusta ovat luennoijan omia päähänpinttymiä. . .)

Logiikka on jo Aristoteleen ajoista (300-luvulta e.a.a.) ollut ”oppi oikeasta päättelystä”: menetelmä jolla kaikkien tunnustamista tosiasioista lähtien voidaan epäilijä vakuuttaa johtopäätöksestä tekemällä sarja vain sellaisia *päättelyaskelia* jotka kaikki myöntävät muodollisesti oikeiksi.

1900-luvulle tultaessa kaivattiin *matematiikalle varmaa perustaa*. Päätettiin *formalisoida looginen päättely*.

Päättely (argumentointi) alun perin *kielellinen* toimitus, ja siksi matemaattinen logiikkakin kielen huomioiva matematiikan haara.

Ongelma: Miten formalisoida ”päättelyaskel jonka kaikki myöntävät muodollisesti oikeaksi”?

Ratkaisu: Formalisoidaan *(mekaanisen) laskettavuuden käsite* eli algoritmit 1930-luvulla – siis ennen tietokoneita annettiin matemaattinen vastine ikivanhalle intuitiiviselle käsitteelle.

Lisätietoja, tuloksia ja tekniikoita Matematiikan laitoksen kurssilla *Matemaattinen logiikka*.

Oma mielipide: logiikka nykyäänkin tärkeää tietojenkäsittelyssä, erityisesti tietokantateoriassa, tekoälyssä, ohjelmointikielten teoriassa ja ohjelmien oikeellisuustarkasteluissa.

Logiikka

Loogisen *objekti*kielen *merkitysoppi (semantiikka)* kertoo miten väitelauseiden *totuus* määritellään.

Tarski: perusväitteiden yhdistelyoperaatiot (”. . . ja. . . ”, ”kaikilla. . . ”,. . .) puretaan induktiivisella määritelmällä *meta*kieleen jossa perusväitteiden totuus ”nähdään helposti”.

”Yks’ vain nainen maailmassa on” kirjoitetaan tavallisella (ensimmäisen kertaluvun predikaatti)logiikalla

$$\exists x.\mathit{nainen}(x) \wedge \forall y.\mathit{nainen}(y) \rightarrow y = x$$

joka kääntyy metakielelle ”on alkio x joka on *nainen* ja jokaisella alkiolla y pätee, että jos y on *nainen*, niin y on sama alkio kuin x”.

Sitten vain katsotaan ulos ikkunasta!

(Logiikka, jatkoa)

Mekaaniset *päättelysäännöt*

$$\frac{\textit{premissi}_1 \ldots \textit{premissi}_k}{\textit{johtopäätös}}[\text{sivuehdot}]$$

valitaan totuuden säilyttäviksi: jos *premissi*t ovat tosia [ja sivuehdot voimassa] niin *johtopäätös*kin on tosi.

Olkoon jo todistettu "jos ϕ niin ψ". Siitä voidaan päätellä "jos on jokin x jolla ϕ niin ψ" mikäli ei aiemmin oletettu että väitteiden ϕ ja ψ alkiot x olisivat samat:

$$\frac{\phi \to \psi}{(\exists x \phi) \to \psi}[x \text{ ei } \psi\text{:ssä}]$$

Päättelystä enemmän Matematiikan laitoksen kurssilla *Logiikka I*.

Entä sääntöjoukon *täydellisyys* eli voidaanko kaikki tosiasiat myös todistaa? 1KL: kyllä.

(Logiikka, jatkoa)

Maailma kuvataan *teorialla*: joukolla tosina pidettyjä loogisia lauseita eli *aksioomia*.

Kysymykseen "onko Ψ totta?" vastataan

1. kääntämällä kysymys loogiseksi kaavaksi ψ,

2. yrittämällä todistaa käännös teoriasta,

3. ja vastaamalla "kyllä" jos todistus *löytyy*.

Tarpeeksi vahvoissa logiikoissa (kuten 1KL) ei mikään todistuksenetsintäalgoritmi voi aina tietää milloin etsintä on tuomittu epäonnistumaan ja voidaan vastata "ei".

Täydellisillä teorioilla ei ole tätä epätietoisuusongelmaa.

Laskenta

Lähtötiedot (kuten luvut) kirjoitetaan sovitulla syötesyntaksilla (kuten numerojonoina). [käännös logiikkaan]

Laskulaite tekee toimiessaan yksikäsitteisesti määriteltyjä perusaskeleita. [päättelysäännöt]

Perusaskelperhe on *universaali* jos niitä yhdistelemällä voi toteuttaa minkä tahansa laskentaprosessin. [sääntöjen täydellisyys]

Vaaditun kuvauksen toteuttavien perusaskeljonojen (ääretön) perhe kuvataan (äärellisellä) ohjelmalla. [teoria]

Annetuilla syötteiden kuvauksilla suoritetaan (ainoa) mahdollinen perusaskelten jono. [todistetaan teoriasta]

Jos jono aina päättyy, on kyseessä algoritmi. [teorian täydellisyys]

(Laskenta, jatkoa)

Jonon päättyessä luetaan vastaus ja käännetään sen syntaksi takaisin merkitykseensä. [on/ei]

Logiikka kuvailee epäsuorasti miten maailman asiat *ovat*.

Laskenta neuvoo miten tästä kuvauksesta voidaan *selvittää* se.

$$\text{algoritmi} = \text{logiikka} + \text{kontrolli}$$

Logiikkaa pidetään ohjelmointia *deklaratiivisempana* koska siinä esitetään vain *mitä* muttei *miten*.

Deklaratiivisiin ohjelmointikieliin on lisätty piirteitä tietämyksen esittämisestä: esim. oliokielissä voidaan ilmaista suoraan määritelmät muotoa "X on sellainen Y jolla. . ."

3 tietä loogisesti perusteltuihin algoritmeihin

1. tie: väittämät

Kalvoilla 48–54 ohjelmoija liitti sopivia väittämiä ohjelmansa – suorituspolkujen kannalta – "strategisiin" kohtiin.

Kalvoilla 55–56 ohjelmoija liitti sopivia konvergentteja ohjelmansa silmukoihin takaamaan niistä poistumisen.

Väittämä- ja konvergentti"nuolet" ovat väitelauseita muotoa "jos ennen. . . niin jälkeen. . . ". Esimerkiksi nuolessa II $\rightarrow$ II puhuttiin muuttujien (X, Y) arvoista ennen (X_1, Y_1) ja jälkeen (X_2, Y_2) tarkasteltavaa laskentapolun pätkää.

Nämä väitelauseet voidaan muotoilla "tavallisessa" logiikassa ja (yrittää) todistaa formaalisti, tai varmistua niiden totuudesta semanttisin järkeilyin.

Näin toiminee jokainen ohjelmoija – ellei muualla niin kommentoidessaan koodiaan.

Algoritmeja voi suunnitella ja ohjelmia kirjoittaa myös ”toisin päin”: ensin annetut alku- ja loppuväittämät, sitten niiden väliin sopiva puolivälin väittämä, ja ”induktiivisesti” samoin alku- ja loppupuoliskon – hajoita ja hallitse!

Itse laskentaoperaatiot kirjoitetaan toteuttamaan ohjelman tilan muutos kahden peräkkäisen väittämän välillä. Silmukoihin ja rekursioon liitetään vastaavat pysähtymiskonvergentit.

Lisätietoa kurssilla *Algoritmien oikeellisuus ja johtaminen*.

Kalvon 28 lomituslajitteluun voitaisiin päästä seuraavasti:

Alku: syötteenä on järjestämätön lista.
Loppu: tuloksena on sama lista järjestettynä.

Alku: syötteenä on järjestämätön lista.
Väli: syöte on jaettu kahteen järjestettyyn noin yhtä pitkään listaan.
Loppu: tuloksena on sama lista järjestettynä.

Alku: syötteenä on järjestämätön lista.
Alkuväli: syöte on jaettu kahteen noin yhtä pitkään listaan.
Väli: syöte on jaettu kahteen järjestettyyn noin yhtä pitkään listaan.
Loppuväli: listat on lomitettu.
Loppu: tuloksena on sama lista järjestettynä.

Alku: syötteenä on järjestämätön lista.
— Jaa syöte kahteen noin yhtä pitkään listaan.
Alkuväli: syöte on jaettu kahteen noin yhtä pitkään listaan.
— Järjestä kumpikin lista rekursiivisesti.
Väli: syöte on jaettu kahteen järjestettyyn noin yhtä pitkään listaan.
— Lomita järjestetut listat keskenään.
Loppuväli: järjestetyt listat on lomitettu keskenään.
— Palauta lopputuloksena lomituksen tulos.
Loppu: tuloksena on sama lista järjestettynä.

Tämän tien (pää)ongelmana on, että laskentapolut jäävät loogisen tarkastelun ulkopuolelle.

2. tie: laskentapolkujen logiikka

Edellinen ongelma voidaan ratkaista määrittelemällä logiikka ei vain alkioille vaan myös poluille.

Tai kielellisesti, ottamalla logiikkaan *aikaa koskevia sanoja* kuten "seuraavassa silmänräpäyksessä" ($\bigcirc$), "aina tästä eteenpäin" ($\Box$),. . .

(Aikalogiikasta lisää Harel sivut 289–291.)

Silloin se, että väittämän II totuus säilyy, voidaan kirjoittaa vaikkapa

$$\Box(\mathrm{II} \rightarrow \bigcirc \mathrm{II})$$

eli "jokaisella suoritusaskeleella pätee, että jos II on nyt totta, niin se on totta myös tätä seuraavalla askeleella".

Silloin kalvojen 48–56 esimerkkialgoritmin osittaisen oikeellisuuden teoria voisi olla vaikka: edellinen väittämän II säilyvyys ja alustuksen aksiooma

$$\mathrm{I} \rightarrow \bigcirc \mathrm{II}.$$

Niistä voidaan (yrittää) todistaa aikalogiikan omilla päättelysäännöillä tai nähdä semanttisesti osittainen oikeellisuus

$$\mathrm{I} \rightarrow \bigcirc \Box (X = \lambda \rightarrow \Box \bigcirc \mathrm{III})$$

eli ”jos lähtöväittämä I taataan, niin toisesta askeleesta alkaen pätee, että jos silmukkaehto joskus laukeaa, niin sen jälkeen III pätee” .

Menestyksekäs *äärellisissä* järjestelmissä: mikroprosessoreissa, protokollissa,. . . Lisää kurssilla *Automaattinen verifiointi*.

Tämän tien (pää)ongelmana on että deklaratiivinen ohjelmoija ei halua kirjoittaa (juuri) mitään laskentapoluista.

3. tie: "loogiset" ohjelmointikielet

Tällä tiellä edellinen ongelma ratkaistaan määrittelemällä algoritmien kuvausnotaation – ohjelmointikielen – semantiikka laitetason sijasta päättelysääntöjen tapaan.

Yksinkertaisen perusaskeleen idea pois:

Logiikkaohjelmointikielissä kuten Prolog (Harel, sivut 70–71) askeleeksi tulee suoraan (rajoitettu) todistusaskel:

$$\frac{\neg a_1 \vee \ldots \vee \neg a_m \qquad b_1 \wedge \ldots \wedge b_n \rightarrow a_1}{\neg b_1 \vee \ldots \vee \neg b_n \vee \neg a_2 \vee \ldots \vee \neg a_m}$$

Funktionaalisissa kielissä kuten Lisp (Harel, sivut 68–70) askeleeksi tulee monimutkaisemman lausekkeen sievennys:

$$\frac{\texttt{if true then } e_1 \texttt{ else } e_2}{e_1}$$

Lisätietoja kurssilla *Symbolinen ohjelmointi*.

Tavoitteena on helpottaa loogisesti oikeiden ohjelmien laadintaa ja todistamista lyhentämällä välimatkaa ohjelmakoodin ja ajattelun välillä.

Kalvon 52 esimerkkialgoritmi voitaisiin tällä tiellä johtaa seuraavasti (vain intuitio):

Annetun merkkijonon S käännös reverse(S) voitaisiin laskea kutsulla rev(S, λ) jos meillä olisi apufunktio rev(X, Y) = reverse(X) + Y.

Johdetaan apufunktio induktiolla sen ensimmäisen (merkkijono)parametrin X suhteen.

Ei-rekursiivisessa perustapauksessa $X = \lambda$ vastaus on Y.

Rekursiivisessa tapauksessa $X \neq \lambda$ on olemassa lyhyempi jono tail(X) ja merkki head(X).

Meillä on siis lupa käyttää lauseketta

$$\text{rev}(\text{tail}(X), \text{head}(X) + Y)$$

koska se on määritelty induktiivisesti.

Se on myös oikea valinta, sillä sen arvo on induktiivisesti

$$\text{reverse}(\text{tail}(X)) + \text{head}(X) + Y.$$

Koska head(X) on merkki, tämä on

$$\text{reverse}(\text{head}(X) + \text{tail}(X)) + Y$$

eli haettu

$$\text{reverse}(X) + Y.$$

Yhteenvetona algoritmiksi saadaan siis

function reverse(S)=rev(S,λ)

missä

function rev(X,Y)=
  if $X = \lambda$ **then**
    Y
  else
    rev(tail(X),head(X)+Y)
  end if.

Algoritmin tekeminen ja oikeaksi todistaminen kulkivat siis käsi kädessä. Ohjelmoijan vastuulle jäi etenemisstrategian valinta.

Tulos on (syntaktisin muutoksin) valmis ohjelma funktionaalisella kielellä.

Lähestymme *konstruktiivista* matematiikkaa, jossa olemassaoloväitteen "on olemassa sellainen x jolla $P(x)$" saa todistaa vain *näyttämällä* jonkin sellaisen sopivan x.

(*Ei* siis esimerkiksi ristiriidalla "jos sellaista x ei olisi, niin siitä seuraisi $0 = 1$".)

Väitteen "jokaisella x on olemassa jokin y jolla $Q(x, y)$" todistus on silloin sellainen *funktio* f joka poimii kullekin x jonkin sopivan $y = f(x)$ jolla $Q(x, f(x))$.

Myös tämä f on näytettävä lausekkeena (eikä vain parijoukkona) – eli annettava sille algoritmi!

Silloin ongelman spesifikaatiota vastaavan olemassaoloväitteen todistus on sen mukainen algoritmi.